Nexus Modeling and Distributed Simulation: A RESTful Framework for Understanding and Predicting Dynamics of Interacting Water and Energy Systems

by

Seyed Mostafa Derakhshandeh Fard

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

Approved December 2022 by the
Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
Michael Barton
Arunabha Sen
Ming Zhao

ARIZONA STATE UNIVERSITY

May 2023

ABSTRACT

Water, energy, and food are essential resources to sustain the development of the society. The Food-Energy-Water Nexus (FEW-Nexus) must account for synergies and trade-offs among these resources. The nexus concept highlights the importance of integrative solutions that secure supplies to meet demands sustainably. The existing frameworks and tools do not focus on formal model composability, a key capability for creating simulations created from separately developed models. The Knowledge Interchange Broker (KIB) approach is used to model the interactions among models to achieve composition flexibility for the FEW-Nexus.

Domain experts generally use the Water Evaluation and Planning (WEAP) and Low Emissions Analysis Platform (LEAP) systems to study water and energy systems, respectively. The food part of FEW systems can be modeled inside the WEAP system. An internal linkage mechanism is available for combining and simulating WEAP and LEAP models. This mechanism is used for the validation and performance evaluation of independent modeling and simulation proposed in this research. The Componentized WEAP and LEAP RESTful frameworks are component-based representations for the legacy and closed-source WEAP and LEAP systems. These modularized systems simplify their use with other simulation frameworks.

This research proposes two interaction model frameworks based on the Knowledge Interchange Broker approach. First, an Algorithmic Interaction Model (Algorithmic-IM) was developed to integrate the WEAP and LEAP models. The Algorithmic-IM model can be defined via programming language and has a fixed cyclic execution protocol. However, this approach has tightly interwoven the interaction model with its execution and has limited support for flexibly creating model hierarchies. To overcome these restrictions, the system-theoretic Parallel DEVS formalism is used to develop a DEVS-Based Interaction Model (DEVS-IM). As in the Algorithmic-IM,

the DEVS-IM is implemented as a RESTful framework, uses MongoDB for defining structural DEVS models, and supports automatic code generation for the DEVS-Suite simulator. The DEVS-IM offers modular, hierarchical structural modeling, reusability, flexibility, and maintainability for integrating disparate systems.

The Phoenix Active Management Area (AMA) is used to demonstrate the real-world application of the proposed research. Furthermore, the correctness and performance of the presented frameworks in this research are evaluated using the Phoenix-AMA model.

This dissertation is dedicated to:

***my parents**, for their unconditional love and support*

***my wife**, for her tremendous encouragement, understanding, and love*

*and **my brothers and sister**, who were true friends in my entire life*

ACKNOWLEDGMENTS

First and foremost, I am extremely grateful to Prof. Hessam Sarjoughian for his invaluable advice, continuous support, and patience during my Ph.D. study and research. His immense knowledge and plentiful experience have encouraged me throughout my academic study and daily life. Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Michael Barton, Prof. Arunabha Sen, and Prof. Ming Zhao, for their insightful comments and encouragement.

Next, I acknowledge members of the ACIMS, present, and past, for their friendship, assistance, and support. I specifically thank Soroosh Gholami, Abdurrahman Alshareef, Chao Zhang, William Boyd, Masudul Quraishi, Moon Gi Seok, Xuanli Lin, Rohit Sinha, Sheetal Chandrakant Mohite, and Forouzan Fallah.

Furthermore, I am fortunate to have been a part of the InFEWs project. This research is based upon work primarily supported by the National Science Foundation (NSF) under NSF Grant #CNS-1639227. I appreciate the collaboration with Prof. Ross Maciejewski, Prof. Dave White, Prof. Rimjhim Aggarwal, Prof. Giuseppe Mascaro, Prof. David Sampson, Dr. Adil Mounir, Dr. Xin Guan, Dr. Leah Jones, Dr. Adenike Opejin, Dr. Yuxin Ma, and Mayuri Roy Choudhury.

Moreover, I extremely appreciate the help of the administrative staff, especially Christina Sebring, Pamela Dunn, and Monica Dugan, whom I troubled a lot but were always there to help me.

Lastly, my family deserves endless gratitude: my father for being the role model of my life, my mother for unwavering support, and my brothers and sister for always believing in me. Without you, I would not be the person I am today. I would like to thank my wife, *Elmira*, for her love, constant support, and for keeping me sane. But most of all, thank you for being my best friend.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Page

Table Page

LIST OF FIGURES

Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

LIST OF LISTINGS

Page

Chapter 1

INTRODUCTION

Recently, there has been an increasing focus on the dynamics, interactions, and feedback between water, energy, and food systems. By 2030, it is expected for water demand to increase by 12% (Boretti & Rosa, 2019), energy by 14% (Cozzi et al., 2020), and food demands by 12% (Serraj & Pingali, 2018) concerning the 2019 levels, mainly because of the increase in population, urbanization, climate change (Gondhalekar & Ramsauer, 2017; Islam & Karim, 2019), and resource scarcity (Ma et al., 2018; Xia & Yan, 2022). These resources are often managed separately, although policymakers and resource managers need to understand the interactions among them. Understanding the Food-Energy-Water Nexus (FEW-Nexus) is necessary for better use, production, and management (Hoff, 2011; Keairns et al., 2016). The "nexus" refers to the relationships that connect food, energy, and water systems to one another. The "incorporation" and "cross-linking" defining the nexus is crucial for understanding and managing the food, energy, and water systems as a whole system. Improving our understanding of these systems, as a whole system, requires modeling the degree to which each system depends on and affects others (see Figure 1.1). It is essential for making informed and mutually compatible decisions for short-/long-term planning.

Combined water, energy, and food resource planning is receiving more and more attention from varied stakeholders, including academia, public and local national, and international organizations. Thus, it is no surprise that an in-depth understanding of the FEW-Nexus is crucial for sustainable resource planning. As shown in

Figure 1.1, the FEW-Nexus system has bi-directional dependencies. For example, water is needed to irrigate land for food production and cool power plants for energy generation. Energy is required for agricultural practices, transport, and treat water. Agricultural activities can generate energy through biofuels but also affect water quality. In addition to these internal interactions between the systems, other external factors (e.g., climate change, population growth, and economic instabilities) increase the complexity of the whole system. For example, what will happen in the nexus if the price of energy increases by a resource producer? Thus, constraints in one area can significantly impact others. Such inter-connectivity aims for cross-sector coordination rather than sector-specific optima to avoid unintended side-effects and negative sectoral trade-offs. Knowledge of the linkages, synergies, and conflicts in the FEW-Nexus model is needed to provide decision-making mechanisms for policymakers in each resource most likely to produce positive effects in the other resources.

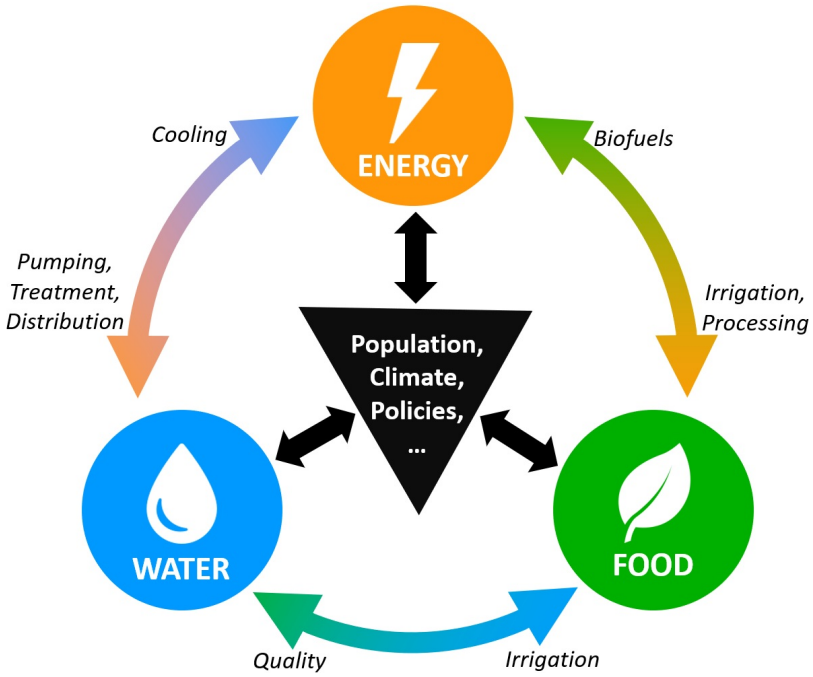


Figure 1.1: Conceptual Diagram of the FEW-Nexus with Exemplar Couplings.

Numerous and diverse tools/frameworks have been presented in different contexts since the interpretation and centrality of the FEW-Nexus. Those tools/frameworks can be roughly categorized into; 1) Resource-Environmental Footprint Quantification Model: to quantify the resource and economic efficiency (e.g., UWOT (Baki & Makropoulos, 2014) and REWSS (A. T. Dale & Bilec, 2014)), 2) Assessment and Systematic Simulation Model: to assess and model the performance of the FEW systems (e.g., WEF Nexus Tool 2 (Daher & Mohtar, 2015) and CMDP (Nanduri & Saavedra-Antolnez, 2013)), and 3) Optimal Management / Integrated Model: to consider multiple systems and the interactions between them as an intricate process involving several modeling and computational complexities (e.g., CLEWS (Howells et al., 2013) and WEAP-LEAP (SEI, 2022b; Sieber et al., 2005)). The research in this article is in the context of Integrated Models. The metropolitan Phoenix serves as a subject study area for this research on the FEW-Nexus.

Regarding modeling disparate systems, examining systems in terms of their parts and relationships allows some parts of a system-of-systems to be modeled in detail while all other parts (that affect or are affected by it) are simple or even excluded. This is attractive as system complexity and scale can be significantly constrained by replacing the dynamics of a system as inputs and outputs. For example, in modeling a water system that uses solar energy, the amount of available photovoltaic energy can be modeled as piecewise input regimes. A key consequence of this choice is that the input regime is non-functional. In contrast, a reactive model produces outputs in part based on consuming inputs dynamically from other models. This photovoltaic model supplies energy to the water system subject to water demand fluctuation can lead to a better understanding of a water system that cannot be achieved through input data alone. From this vantage point, the need for component-based modeling

and simulation is evident for understanding the interactions among the different parts of a complex system (e.g., the FEW-Nexus) (Hoff, 2011; Keairns et al., 2016).

Considering the water, energy, and food systems as separate models and coupling them via a third model brings up not only the modularity for the systems but also flexibility and rigorous predictive simulation for the integrated FEW-Nexus (Fard & Sarjoughian, 2020, 2021b). Therefore, at least four models can be considered to represent the whole system; one for the water system, one for the energy system, one for the food system, and one for their nexus. From a highly abstract modular perspective, water, energy, and food systems have some input/output ports to interchange data from/to the other system. In a general view, systems can have different model structures and behaviors defined according to formal specifications. This research is based on using the Water Evaluation and Planning (WEAP) system (SEI, 2022d) for the water and food sections and the Low Emission Analysis Platform (LEAP) system (SEI, 2022b) for the energy section. The advantage of relying on previously established frameworks can be reducing the effort and resources needed for model development. In addition, domain experts can participate in collective work using existing tools and reduce learning curves. It also allows for better use of previously acquired knowledge and experience.

1.1 Motivation & Problem Statement

The nexus concept initially emphasized the intricate nature of the food, energy, and water sectors that needed to be addressed together (Bazilian et al., 2011). The World Economic Forum (Initiative et al., 2012), as the earliest organization, advocated paying more attention to the separable linkages among the three sectors. Subsequently, many related studies have been undertaken (Biggs et al., 2015; Siddiqi & Anadon, 2011). Modeling the FEW-Nexus is a complex and challenging task that

requires extensive data on specific study areas (Kaddoura & El Khatib, 2017). Since presenting the FEW-Nexus, although numerous and diverse frameworks/tools have been used in different contexts, there is currently no modeling framework with the instruments needed to facilitate comprehensive analyses of the FEW-Nexus (P. Zhang et al., 2019).

The first challenge is understanding each model (i.e., water, energy, and food) individually. Then, due to the complex interdependent nature of the FEW-Nexus, it is often unclear how an action in one sector may impact the others. So, a challenge is extracting the interconnection between food, energy, and water models. The "nexus" refers to the complex and inherent linkages among models. To better grasp the interdependency among FEW sectors, the system must be studied and modeled at different levels (Giampietro et al., 2013). So, another challenge is defining the objectives and the level of abstraction. For example, the WEF Nexus Tool 2.0 is a web-based tool for guiding resource allocation at the country level, for a given level of food self-sufficiency and a set of technologies, land uses, and resource availabilities (Daher & Mohtar, 2015); and the Climate, Land, Energy, Water Climate, Land, Energy, Water (CLEW) framework is based on a system's thinking approach to analyze interactions between interconnected sectors (Kaddoura & El Khatib, 2017). Tools like WEF Nexus Tool 2.0 have higher abstraction than tools like CLEW, which are defined by their significant data requirements and resource intensity. The complexity of an integrated modeling framework could make its development and application both difficult and costly. The data intensity brings up the next challenge, which is the lack of accurate data/information. In other words, one of the most significant limitations for the FEW-Nexus modeling is the extensive amount of required data (Kaddoura & El Khatib, 2017).

1.1.1 Food-Energy-Water Nexus Tools/Frameworks

Considering simulation studies of integrated food, energy, and water systems, frameworks/tools such as Precipitation Runoff Modeling System (PRMS) (Markstrom et al., 2015) and WEF-Nexus Tool 2.0 (Daher & Mohtar, 2015) have been developed. These tools are not based on component-based modeling principles and service-oriented computing. The PRMS is a deterministic, distributed-parameter, physical process-based modeling system developed to evaluate the impacts of various climate and land use combinations on surface-water runoff, sediment yields, and general basin hydrology. The WEF Nexus Tool 2.0 is a scenario-based tool for guiding resource allocation at the country level for a given level of food self-sufficiency and a set of technologies, land uses, and resource availabilities. The CLEW framework is based on a system's approach to analyzing interactions between interconnected sectors (Kaddoura & El Khatib, 2017). It uses existing simulation tools (WEAP, LEAP, and AEZ) based on a modular structure to illustrate synergies and trade-offs within the CLEW areas for decision-making related to achieving development goals (Howells et al., 2013). The CLEW framework applies to different geographical scales from global to regional, national, and urban levels ("CLEWS-Home", 2022). Tools with higher complexity scores (e.g., CLEW, WEAP, and LEAP), unlike those with lower complexity scores (e.g., WEF Nexus Tool 2.0), can capture details for specific resource interactions, whereas they are unable to cover a larger number of interactions and system components, simultaneously (Dargin et al., 2019).

1.1.2 WEAP and LEAP Componentization

In a general view, the advantage of relying on previously established tools and frameworks (instead of developing from scratch) is reducing the effort and resources

needed for model development. Also, domain experts can participate in collective work using existing tools and better use previously acquired knowledge and experience. Some popular modeling and simulation tools and frameworks, such as the WEAP and LEAP systems, appear to be component-based since non-componentized models are displayed as components in the tool. As shown in Figure 2, the WEAP models are defined as a network of water supply and demand entities (nodes) that are connected via transportation entities (links). Likewise, the LEAP models are defined as a network of resource and demand entities (nodes) that are connected via transformation entities (links). Indeed, the WEAP and LEAP models are based on mass-balanced equations and shared variables. Approaches that use shared variables amongst models lack the flexibility afforded by component-based modeling frameworks. Simulations developed using component-based modeling approaches are essential in detailing different behaviors belonging to different parts of a system-of-systems. In a component-based modeling framework, each model is a standalone component having its inputs, outputs, and functions encapsulated. So, it promotes modularity, which is a crucial enabler for synthesizing hierarchical models (Zeigler et al., 2017).

A particular consequence of modeling frameworks such as WEAP and LEAP is the difficulty of using them systematically with other frameworks. A desirable modeling framework should simplify and promote combing models that are developed in different frameworks or tools. In such a modeling framework, each model entity is a standalone component having its inputs and outputs, and functions encapsulated and thus not shared with any other model. In a component-based modeling framework, models are clients and servers that can be independently acted on. In a direct way, the componentization of tools such as WEAP and LEAP can further their use in modeling and simulation system-of-systems, including the class of the FEW system (Hoff, 2011).

The composition of models is considered essential in developing complex systems and simulation models capable of expressing a system's structure and behavior (Sarjoughian, 2006). From a component-based modeling approach, systems are integrated from homogeneous or heterogeneous sub-systems. Each sub-system can be considered an independent system or component. A component is an encapsulated software unit with a known set of inputs and expected output behavior, where the implementation details may be hidden or unknown (Petty et al., 2014). Considering the water, energy, and food systems as separate models/components and coupling them via an interaction model leads to modularity and composability for modeling, simulating, and evaluating the FEW and its nexus (Fard & Sarjoughian, 2020; Fard et al., 2020). The components of the water and energy models and the defined relations in the interaction model can be considered as a network (see Figure 1.2).

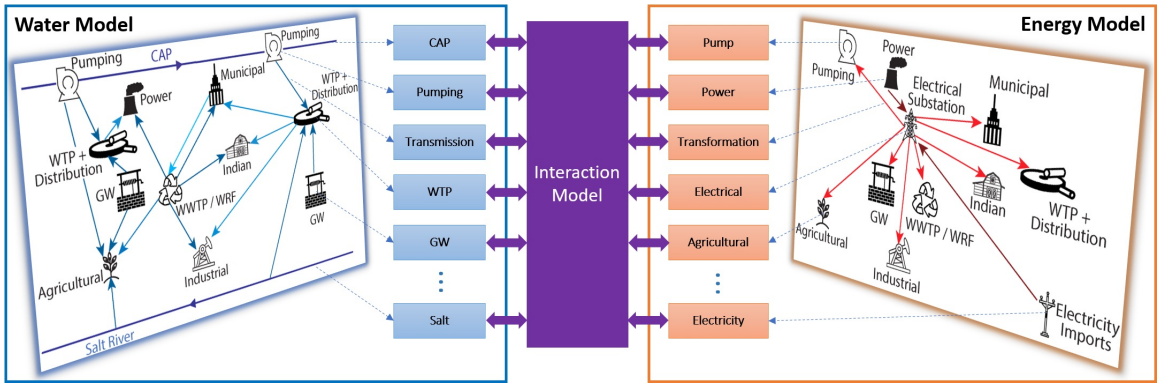


Figure 1.2: Conceptual Diagram of Componentized Water and Energy Network Models.

1.1.3 Composability

Creating a large and complex system simulation requires breaking the problem into parts that can be addressed separately. Also, understanding complex systems requires decomposition because no one can otherwise comprehend the full details. Furthermore, testing systems are vastly simplified if one can do it module by module

and then at the system level (Davis & Anderson, 2004). Composition, as a verb, is the process or capability of selecting and assembling components for execution. Composition, as a noun, refers to a set of components composed to produce an integrated or interoperable whole (Petty et al., 2014). Composability is a system design principle that deals with the inter-relationships of components. A highly composable system provides components that can be selected and assembled in various combinations to satisfy specific user requirements and answer trade-off questions (Davis & Anderson, 2004). The main challenge of achieving model composability has appropriate concepts and methods to compose different model types with well-defined syntax and semantics (Sarjoughian, 2006) and its use for specific application domains (Mayer & Sarjoughian, 2007).

Component-based approaches decompose complex designs into more manageable pieces and help reuse existing modules. Creating a simulation of a large, complex system requires breaking the problem into parts that can be addressed separately. This helps to permit specialization, facilitate computing alternative ways of handling a given component, and maintain the software over time. Furthermore, new components in a component-based approach can be created via the hierarchical composition of the previously proven components. Thus, the interaction styles among components are well-defined at each level and are constrained to that local scope. So, the first requirement is having a component-based approach for modeling the system. The modeler should be able to assemble the suitable system-of-systems model with plug-and-play and answer trade-off questions. On the other hand, understanding complex systems requires decomposition because no one can comprehend the whole's details. Testing systems are vastly simplified if one can do it module by module and then at the system level (Davis & Anderson, 2004). So, modularity is good, but composability is more than modularity.

1.1.4 Interaction Model Specification

The FEW-Nexus is inherently complex. It requires the integration of complex physical phenomena and built systems. This complexity is further compounded by how scholars, policymakers, and other stakeholders engage with the concept and their different perspectives to understand it. A comprehensive review is reported five perspectives to view the FEW-Nexus (Proctor et al., 2021):

- Ecosystem Health Perspective: A significant emphasis is placed on quantifying environmental impacts and encouraging sustainable practices. The primary goal from this perspective is to manage FEW sectors such that environmental impacts are minimized while still achieving FEW securities. Life Cycle Assessment (Roy et al., 2009) and Ecological Indicators (Saladini et al., 2018) are two of the most common tools used when viewing the nexus from this perspective.

- Waste Management Perspective: Emphasizes the critical role of waste in resource security. Wasted food represents a significant source of inefficiency within Nexus. Water is directly wasted through excessive water use in irrigating crops and domestic areas, such as lawns and excess household use. One increasingly used framework for understanding waste management is the Circular Economy (Geissdoerfer et al., 2017).

- Institutional Change Perspective: This focuses on institutions' actions, studying how policies have previously impacted the Nexus and are currently impacting it while also identifying methods for implementing this understanding to recommend beneficial strategies. Many of the studies focusing on the perspective start with a quantitative analysis such as Input-Output analysis. Others use

an agent-based modeling approach to try and predict how stakeholders and ecosystems may react to changes in the FEW-Nexus policies.

- Stakeholder Trust Perspective: Explores public opinions of the Nexus. This work directly studies current opinions towards the Nexus via stakeholder engagement. In addition, it predicts how opinions may shift in the future. One of the most common tools used to assess citizens' views on the nexus is to survey them.

- Learning Process Perspective: The learning process of stakeholders, why and how they learn, has been explored from multiple directions, e.g., economics and political science point of view, using a psychological lens, and adopting behavioral and cultural traits (across multiple generations). The learning process perspective seeks to understand how public opinions towards the FEW-Nexus develop over time and how outside forces shape this understanding.

Overall, endeavors in the FEW-Nexus domain primarily focus on three aspects: (1) understanding and interpreting the concept of the FEW-Nexus; (2) developing modeling approaches that can identify trade-offs and synergies of the FEW system, internalize social and environmental impacts, and guide the development of cross-sectional policies; and (3) conducting empirical research to show the driving forces of the FEW system and probe collaboration potentials between different stakeholders (P. Zhang et al., 2019).

The Knowledge Interchange Broker (KIB) approach has been introduced to formalize the interactions between the models specified in different modeling formalisms (Sarjoughian, 2006). The KIB approach can be used to define *data mappings*, *synchronization*, *concurrency*, and *timing*. The conceptual basis of the KIB is that disparities between different syntaxes and semantics need to be accounted for with a separate

model syntax and semantics, thus enabling independent modeling of interactions between the composed models. This approach has been applied to different domains (Barton et al., 2016; Huang et al., 2009). In this research, the KIB concept is used to define the relationship between the WEAP and LEAP models externally.

Using a formal modeling method to model and simulate the interactions between disparate models is advantageous. A component-based, hierarchical modeling approach that aligns with system thinking helps with the development, reuse, and maintainability of interaction models. A discrete-event simulation method is used in this research to define a framework specification based on the KIB approach to defining the interaction between disparate systems. In a discrete-event abstraction, only a finite number of relevant events occur during a bounded time interval. This contrasts with continuous models, which take continuous signals as input. A discrete-event system is characterized by the output events which are generated in response to a series of input events Zeigler et al., 2018. The Parallel Discrete EVent System (DEVS) specification is a popular formalism for modeling complex dynamic systems using a discrete-event abstraction. Formalism has a rigorous formal definition and strong support for modularity (i.e., models can be hierarchically nested). An Interaction Model framework is developed in this article based on the KIB approach and Parallel DEVS formalism (Fard & Sarjoughian, 2021b).

1.2 Research Objective

The needs mentioned in Section 1.1 motivated us toward studying the WEAP and LEAP systems to understand their modeling approaches, execution and calculation processes, and accessible input and output data. In the next step, there is a need to define a well-defined componentized specification (componentization) for the WEAP and LEAP legacy systems. It facilitates integrating the WEAP and LEAP systems

(legacy modeling and simulation systems) with other modeling and simulation tools/frameworks to model and simulate systems-of-systems (i.e., the FEW-Nexus). The componentization requires exposing the entities of water and energy models in the WEAP and LEAP systems to be represented as external logical components to be combined with models of other systems. The preliminary work on this research described a webservice-based design and implementation using the idea of software components for the WEAP and LEAP systems (Fard & Sarjoughian, 2019).

After having a component-based representation for the WEAP and LEAP systems, the most significant section is the design and specification for the Interaction Model (see Figure 2). The Interaction Model has some inputs and outputs connected to the external systems (i.e., the WEAP and LEAP systems). Generally, the external systems connected to the Interaction Model can have arbitrary system specifications (even though the WEAP and LEAP have the same specification). The Interaction Model can receive input data, process them, and send the results to the external systems. Furthermore, the constraints and validations can be applied to the Interaction Model. The execution of the whole system (the Interaction Model and the connected systems) is also controlled in the Interaction Model. Thus, our next research question is how to model and simulate the nexus of the Componentized WEAP and LEAP systems.

The WEAP and LEAP systems can bi-directionally share data (called WEAP-LEAP internal linkage). On the one hand, the WEAP and LEAP source codes are not public, so their internal linkage cannot be examined/evaluated. On the other hand, the WEAP and LEAP models must obey the internal linkage constraints (e.g., having the same time interval and time granularity in a year and manual execution). So, the WEAP-LEAP internal linkage cannot be considered a flexible modeling methodology for the FEW-Nexus, due to mentioned limitations (the detail of the WEAP-LEAP

internal linkage is presented in Section 3.3). But the WEAP-LEAP internal linkage is used in our research to validate the correctness of the modeling and simulation in the Interaction Model.

The overarching goal of this research is to develop a service-oriented framework for developing simulations targeted at a scientific understanding of FEW dynamics. The tool is based on the Software-as-a-Service (SaaS) model. Cloud providers manage the infrastructure and platforms that run the water, energy, food, and interaction model applications, a capability foreseen to be essential for large-scale, recurrent FEW-Nexus studies. The tool will offer composable modeling and interoperable simulation with features for data management infrastructure. The aim is to enable a better understanding of the FEW-Nexus at decision-relevant temporal. Such a service-oriented modeling and simulation tool can benefit the creation of verified and validated simulation models serving the needs of resource managers and decision-makers responsible for making policies and decisions toward sustainable water, energy, and food resources.

1.3 Contributions

Regarding the mentioned problems in Section 1.1 and the defined objectives in Section 1.2, the main contributions of this research are:

- **Legacy Tools Componentization:** New insight is gained into the development of the Componentized WEAP/LEAP RESTful framework by examining the underlying details and formulation of the WEAP/LEAP entities as proxy component models using the Model Driven Architecture (MDA) approach and the UML diagrams (moving between different levels of abstraction).

- **Algorithmic Interaction Model:** Developed an Algorithmic interaction model framework based on the Knowledge Interchange Broker (KIB) modeling ap-

proach and object-oriented principles which has a minimal number of elements to define the model and a cyclic and synchronous fixed execution protocol.

- **DEVS-Based Interaction Model:** Developed a framework that conforms to the KIB approach and Parallel DEVS formalism. The framework has a set of predefined modeling elements to simplify interaction modeling (for the modelers who are not experts in the DEVS formalism) and define a generic ontology for the disparate external systems connected to the interaction model (family of models is persistently stored in the MongoDB database). Furthermore, the DEVS-Suite simulator is used to develop, test, debug, and run the interaction model.

- **Webservice-Oriented Modeling & Simulation:** Developed a distributed environment for modeling and simulating the Componentized WEAP, Componentized LEAP, and the Algorithmic/DEVS interaction models based on the Service-Oriented Architecture.

- **Water-Energy Nexus Modeling of the Phoenix AMA Model:** Developed the Water-Energy Nexus model using the Algorithmic-IM and DEVS-IM frameworks for the Phoenix Active Management Area. These models replicate the same nexus developed using the WEAP-LEAP Internal Linkage. The developed models using the frameworks are evaluated and validated based on the WEAP-LEAP Internal Linkage model.

Chapter 2

BACKGROUND

The WEAP and LEAP are the primary tools to model the Water and Energy systems in this research. So, the characteristics and capabilities of the WEAP and LEAP systems are described with a focus on their use with other frameworks and tools. Also, their internal linkage is described in detail. Then, the RESTful framework, as the underlying framework for developing the Componentized WEAP and Componentized LEAP, is described. Finally, the Discrete EVent System Specification (DEVS) is presented. DEVS is the infrastructure concept to design the second generation of the Interaction Model.

2.1 Composability

Composition of water and energy models for the FEW-Nexus is the main topic in this research, and the focus is to use the Knowledge Interchange Broker Approach (Sarjoughian, 2006) for interaction modeling. Creating a simulation of a large, complex system requires breaking the problem down into parts that can be addressed separately, it helps to reduce the effects of interruption, to permit specialization, to facilitate computing alternative ways of handling a given component, to maintain the software over time, and to reduce risk by relying upon previously proven components where possible. So, the first requirement is having a component-based approach for modeling the system. The modeler should assemble the suitable system-of-systems model with a plug-and-play approach and answer trade-off questions. On the other hand, understanding complex systems requires decomposition because no one can otherwise comprehend the whole's details. Testing systems are vastly simplified if

one can do it module by module and then at the system level (Davis & Anderson, 2004). So, modularity is good, but composability is more than modularity.

Composition (as a verb) is the process or capability of selecting and assembling components for execution. Composition (as a noun) refers to a set of components that have been composed to produce an integrated or inter-operable whole (Petty et al., 2014). In other words, composability can select and assemble simulation components in various combinations into sound simulation systems to satisfy specific user requirements (Davis & Anderson, 2004). The main challenge of achieving model composability has appropriate concepts and methods to compose different model types. Both the disparate models and their interactions should apply a well-defined syntax and semantics (Sarjoughian, 2006) for given application domains (e.g., (Mayer, 2009)).

Heterogeneous model types (composting models with different model specifications) can offer greater flexibility than homogeneous model types (composting models with the same model specification). Decomposition and composition of models are challenging when models are heterogeneous in terms of consistency of the model specifications (correctness and validation). Consider injecting data and control from an external source that does not have the same approach to model specification and execution. For instance, one model may have an innate concept of time, while the other does not.

Model composition from a system-theoretic standpoint can be classified into *mono-*, *super-*, *meta-*, and *poly-formalism* methods (Sarjoughian, 2006). The first two are grounded in the concept that, in some cases, a single formalism is well suited for modeling different parts of a system. In contrast, the latter two are based on disparate modeling formalisms to describe the parts of a complex system.

In the *mono-formalism* approach, different parts of a system can be modeled via a single formalism (homogeneous models). It is common to use a single modeling formalism to specify one aspect of a system. Using this modeling approach, models of different system parts adhere to a single structure and behavior specification. As shown in Figure 2.1a, $M_{A\cup B,\{\Psi\}}$ is used to model system A (e.g., discrete-event process), system B (e.g., event-based control), and their interactions using modeling formalism Ψ.

In the *super-formalism* approach (see Figure 2.1b), a modeling formalism, $M_{\widetilde{B},\{\Phi\}}$, encapsulates within the super-formalism, $M_{B,\{\Psi\}}$. Models $\widetilde{B}$ and B are different and the former must be encapsulated inside the latter (model B wrappers model $\widetilde{B}$). A general approach is to use multiple model specification abstractions and hide the details of an encapsulated lower-level model specification inside an enclosing higher-level model specification. For example, a simple optimization model can be encapsulated as an I/O System (i.e., atomic) DEVS component. A super modeling formalism can support model specifications that are at the same level of abstraction.

In the *meta-formalism* approach, two models described in different modeling formalisms can be transformed entirely or partially to other formalisms. As depicted in Figure 2.1c, $M_{\widehat{A}\cup\widehat{B},\{\Theta\}}$ is a composition of models with their interactions specified in formalisms Θ. Here models A and B (specified in Ψ and Φ) are mapped to $\widehat{A}$ and $\widehat{B}$. This requires Ψ and Φ transform to Θ. Indeed, the interactions between models A and B are specified in terms of $\widehat{A}$ and $\widehat{B}$. A good example based on this approach is High-Level Architecture (HLA).

In the *poly-formalism* approach, a separate model handles the differences between the composed models. As shown in Figure 2.1d, $M_{A\cup B\cup C,\{\Omega\}}$ is the composition of $M_{A,\{\Psi\}}$ and $M_{B,\{\Phi\}}$, using $M_{C,\{\Omega\}}$. The interactions between models A and B are specified in Ω. Unlike the meta-modeling formalism approach, in the poly model

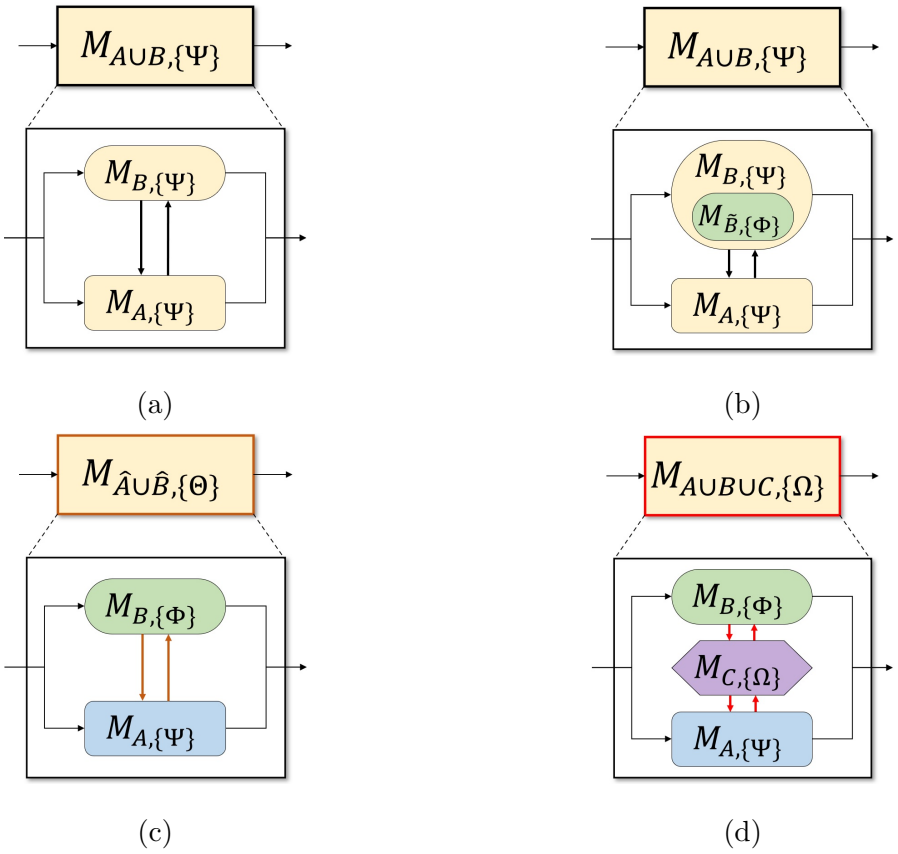


Figure 2.1: Model Composability Approaches. (a) Mono (b) Super (c) Meta (d) Poly Approach.

composability approach, models are not transformed to a set of models, all of which are described following a single modeling formalism. Furthermore, poly modeling formalism is distinct from the strong form of super formalism that can both support the interactions (data and control exchanges) among different model types (e.g., discrete-event and linear optimization) and support different kinds of data transformation and control schemes that are described external to the models that are composed. Using the poly model composability approach, standard forms of data transformation (i.e., aggregation and dis-aggregation) are inherently supported. The best example based on this approach is Knowledge Interchange Broker (KIB).

This research is based on the *poly-formalism* approach (see Figure 2.1d), even though the WEAP and LEAP systems are based on the same formalism. Because our approach is to present a solution as dynamically as possible, for example, to replace the current tools (WEAP and/or LEAP), or add a new framework (e.g., for the Food system, or any new system such as Climate and Economy).

2.1.1 Knowledge Interchange Broker

Using multiple modeling formalisms to model a large system, such as the FEW-Nexus, is crucial. The KIB approach has been introduced to formalize the interactions between the models specified in different modeling formalisms (Sarjoughian, 2006). The conceptual basis of the KIB is that disparities between different syntaxes and semantics need to be accounted for with a separate model syntax and semantics, thus enabling independent modeling of interactions between the composed models. Syntactic characterization specifies the elements (and their relationships) of valid models, whereas semantic characterization pertains to the dynamic behavior of each model in particular formalism. The KIB composition specification is treated as an independent model between the disparate models that explicitly addresses the interaction activities in terms of data transformation, concurrency, synchronization, and timing properties (which account for both structural and behavioral compositions). In other words, the KIB approach emphasizes on separation of model specification and its execution protocol. This separation makes it possible to treat model composability and execution interoperability differently, given different composition approaches. The KIB approach has been applied to different domains (Boyd & Sarjoughian, 2020; Huang et al., 2009; Mayer, 2009; Sarjoughian et al., 2013; C. Zhang et al., 2020).

The data transformation property of the KIB approach involves data aggregation from a set of data values to one data value (e.g., average, maximum, and minimum),

data dis-aggregation from one data value to a set of data values (e.g., distribution and division), and data conversion from one unit to another unit. The concurrency property needs to specify how simultaneous interaction messages are handled. The execution of individual models controlled by the KIB can be as simple as sequential or as complex as asynchronous. The concurrency property is closely related to the synchronization property, which specifies when interaction messages can be sent or received. The composition specification using the KIB provides a systematic approach for describing interactions among disparate models at the modeling level. Finally, the timing property addresses how time is presented and advanced in the individual formalisms and how timing is mapped between the modeling formalisms.

The KIB approach differs from the generic concept of broker architectural pattern (Buschmann et al., 2007), which has been widely used in the software community. The broker pattern is an architectural pattern that can be used to structure distributed software systems with decoupled components that interact by remote procedure calls. It provides location transparency, re-usability, changeability, and extensibility of components. The main responsibility of a broker component is to facilitate communication (message transferring) between the client and server. The broker has no knowledge of the client or server. The interface provided by the broker component can ensure the syntactic properness of the interaction, but it cannot guarantee semantic correctness since it lacks the semantic knowledge of the messages and operations involved in the interaction (Huang, 2008).

2.2 Parallel Discrete EVent System Specification

Parallel Discrete EVent System Specification (Parallel DEVS) is a popular formalism for modeling complex dynamic systems using a discrete-event abstraction (Zeigler et al., 2018). The term "DEVS" is used in the rest of the paper, referring to the Paral-

lel DEVS formalism. The main advantages of DEVS are its rigorous formal definition and its support for modularity. DEVS is a hierarchical, continuous-time formalism devised for the modeling and simulation of reactive systems. Thus, systems can be modeled as a set of communicating automata in a hierarchical fashion using atomic and coupled DEVS models. The formal specification of the atomic DEVS model is:

$$Atomic =< X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >$$

In this description, the input ports/events, output ports/events, and sequential state set are represented by X, Y, and S. The external transition function is $\delta_{ext} : Q \times X \rightarrow S$ where $Q = \{(s, e) | s \in S,\ 0 \leq e \leq ta(s)\}$. This function is a mapping between the occurrence of a bag of external events on one or more input ports and the sequential state set at any time. The internal transition function, $\delta_{int} : S \rightarrow S$, defines how the model reacts to internal events. The confluent function, $\delta_{con} : Q \times X \rightarrow S$, handles the occurrence of simultaneous internal and external events. The output function $\lambda : S \rightarrow Y$ specifies output generation by mapping the state set to a bag of output events on one or more output ports at any instance of time. Finally, the time advance function, $ta : S \rightarrow \mathbb{R}^{+}_{0,\infty}$, specifies the timing behavior of the system.

Hierarchical structures in DEVS are made possible through coupling input and output ports of atomic/coupled models subject to no direct feedback coupling. Coupled models do not contain state information; they only specify how components are placed and connected in a strict hierarchical tree. The coupled model specification in a P-DEVS model is:

$$Coupled = < X, Y, D, M_d | d \in D, EIC, EOC, IC >$$

In the description, X and Y remain as input and output ports/events. D is the index set (component names) for internal atomic/coupled models. $\{M_d\}$ is the set

of internal atomic/coupled models for which the index set D. Finally, the three sets EIC, EOC, and IC contain a set of external output port couplings, a set of external input port couplings, and a set of internal couplings (for internal couplings between atomic/coupled models within $\{M_d\}$), respectively.

A framework of abstract simulators has been proposed to simulate DEVS models (Zeigler et al., 2018). The framework consists of a hierarchy of simulator objects that mirrors the hierarchical structure of the simulated DEVS model. A DEVS simulator corresponds to each atomic model, and a DEVS coordinator corresponds to each coupled model. The correspondence between the model and the simulator objects is illustrated in Figure 2.2.

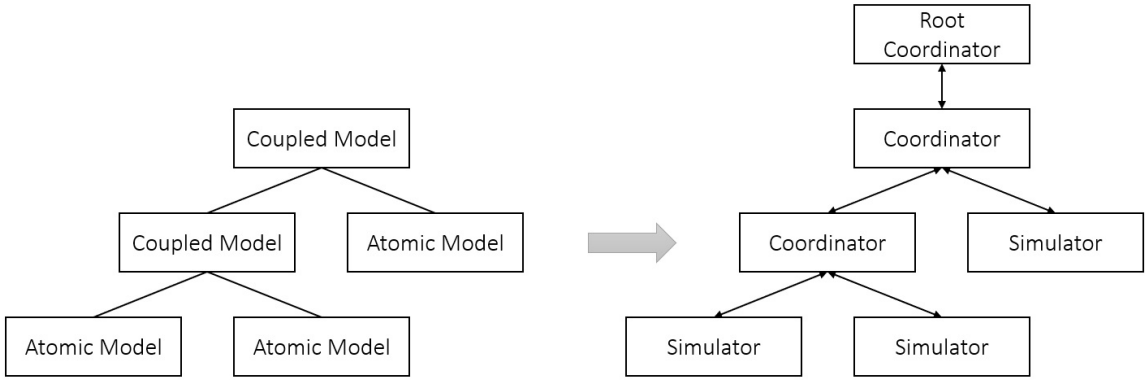


Figure 2.2: Mapping a Hierarchical Model onto a Hierarchical Simulator.

Simulator and coordinator modules manage timing and choice of functions to be executed as well as sending and receiving input/output events for atomic and coupled models they supervise. The control of the simulator module on the timing aspect of an atomic model is depicted in Figure 2.3. The time of the last event (tL), time of next event (tN), and external input events are used by the simulator module to control the execution of an atomic model. Among the internal/external events belonging to all atomic models, the ones with the earliest scheduled time (i.e., tN) will be executed.

Depending on whether an external event, an internal event, or both occur at time instance tN, the appropriate function/s within the atomic model is/are invoked.

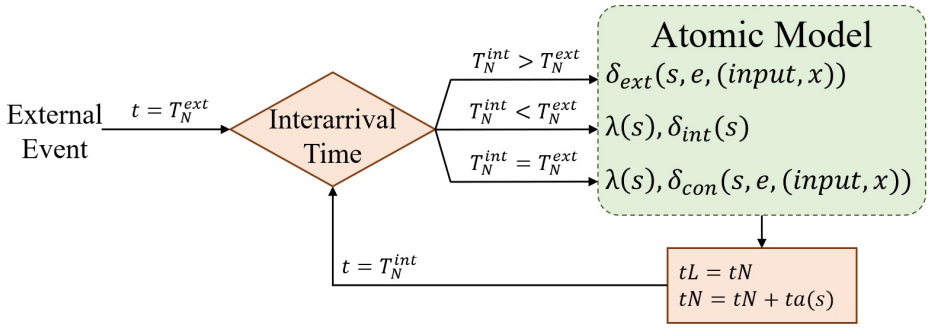


Figure 2.3: Time Management in the Coordinator Module (with Solid Borders and in Orange) and Invocation of the Atomic Model Functions (with Dashed Borders and in Green) in the Simulator Module.

2.2.1 DEVS-Suite Simulator

DEVS-Suite is an object-oriented implementation of the DEVS formalism and an open source, discrete event, and general-purpose simulation environment (ACIMS, 2022c; Kim et al., 2009). It supports describing complex structures and behaviors of systems using object-oriented modeling techniques and advanced features of the Java programming language. The DEVS-Suite user interface provides a consistent, efficient, integrated hierarchical component-based representation of models with run-time I/O and state trajectories and tabular data visualization (ACIMS, 2022d). The DEVS-Suite has a powerful and robust simulation engine that enables the modeler to simulate the flexible application of a model's steps. In the DEVS-Suite, the execution of the models can be animated in terms of the input/output messages for coupled models and the state changes for the atomic models. Every atomic and coupled model component can have its own time-based trajectories and log files for inputs and outputs as well as the common phase and sigma state variables for atomic models (Zengin, 2010). The architecture of the DEVS-Suite simulator environment is Model

Facade View Control (MFVC) (Sarjoughian & Singh, 2004). Simulation data can be displayed with its animation and viewing of time trajectories generated by the DEVS abstract simulator. The DEVS-Suite visualization constraints to visualize the number of the model's ports (restricted to 19 ports), so an atomic or coupled model should not have more than 19 ports to use the visualization part helpfully.

2.3 Service Oriented Architecture

Service-oriented architecture (SOA) is a technology-neutral, platform-independent design, which provides the aspects of reusability, agility, loose coupling, and interoperability with the help of a collection of services. The main advantage of building services is that they provide a standard way of interaction. Nowadays, there are two main web-service protocols. One is the Simple Object Access Protocol (SOAP) (Box et al., 2000), and the other is the REpresentational State Transfer (REST) (Fielding & Taylor, 2002; Fielding, 2000). The former is an XML-based standard communication specification over a particular protocol such as HTTP and SMTP (Simple Mail Transfer Protocol). The latter is a web-based architectural style with flexible message formats such as XML and JSON. Following are the principles that must be satisfied if an interface needs to be referred to as RESTful (Pautasso & Wilde, 2010).

1. **Client-Server**: It separates the user interface implementation from the data storage concerns. It also simplifies the server components to improve the user interface across multiple platforms and higher scalability.

2. **Uniform Interface**: By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified, and the visibility of interactions is improved.

3. **Stateless**: Each client request to the server must contain all information necessary to understand the request. Indeed, the session state is kept entirely on the client.

4. **Resource Caching**: Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

From the client's perspective, the SOAP is based on the operation/method, whereas the REST is based on the resource. The RESTful framework is an implementation of the REST architecture based on the HTTP protocol (Richardson & Ruby, 2008). Asynchronous requests, higher security and reliability, and error reduction are the main reasons for choosing the SOAP standard. Greater scalability, compatibility, performance, and simplicity are the common reasons for choosing the REST standard.

Web services such as the Extensible Modeling and Simulation Framework (XMSF) (Brutzman et al., 2002) are defined as an integrable set of standards, profiles, and recommended practices for web-based modeling & simulation (Sonntag et al., 2011). The XMSF supports the migration of legacy components into web-enabled components for distributed heterogeneous simulation applications. It is based on SOAP and XML (Brutzman et al., 2002). In contrast, the proposed WEAP web-service system benefits from the RESTful web services and the Ecore modeling framework.

2.4 Used Tools to Model the Water and Energy Systems

There exist a wide variety of many tools for modeling and simulating water and energy systems, serving purposes ranging from natural processes to engineered distri-

bution networks. The Water Evaluation and Planning (WEAP) and Low Emissions Analysis Platform (LEAP) systems are used in this research to model and simulate the water and energy systems, respectively. The WEAP and LEAP tools are described in detail in the following sections.

2.4.1 Water Evaluation and Planning System

The Water Evaluation and Planning (WEAP) system is a tool created in 1988 for modeling, simulating, and evaluating water systems (Sieber et al., 2005). Models are defined as a network of water supply and demand entities (nodes) that are connected via transportation entities (links). A WEAP model defines the allocation of water from different sources through preferences and mass balance constraints. The WEAP system provides a set of entities and procedures to study and find solutions to the problems faced by decision-makers. Using a scenario-based approach, each study area has natural watersheds, reservoirs, streams, and canals that serve to supply demands by various users, including households, industry, and agriculture (Yates et al., 2005). The WEAP tool is widely used globally for water allocation and water management (Amin et al., 2018; J. Gao et al., 2017; Höllermann et al., 2010; Lévite et al., 2003; Psomas et al., 2016).

The development of a WEAP model includes several steps (SEI, 2022d). A study is defined to have a time frame, a spatial boundary (see Figure 2.4), system entities, and configuration. The predefined "Current Accounts" scenario, which can be used to calibrate a model, provides a snapshot of actual water demand, pollution loads, resources, and supplies for the system at the initial state. Scenarios are defined using the "Current Accounts" and explore the impact of alternative assumptions or policies on future water availability and use. Finally, the scenarios are evaluated regarding

water sufficiency, costs and benefits, compatibility with environmental targets, and sensitivity to uncertainty in key variables.

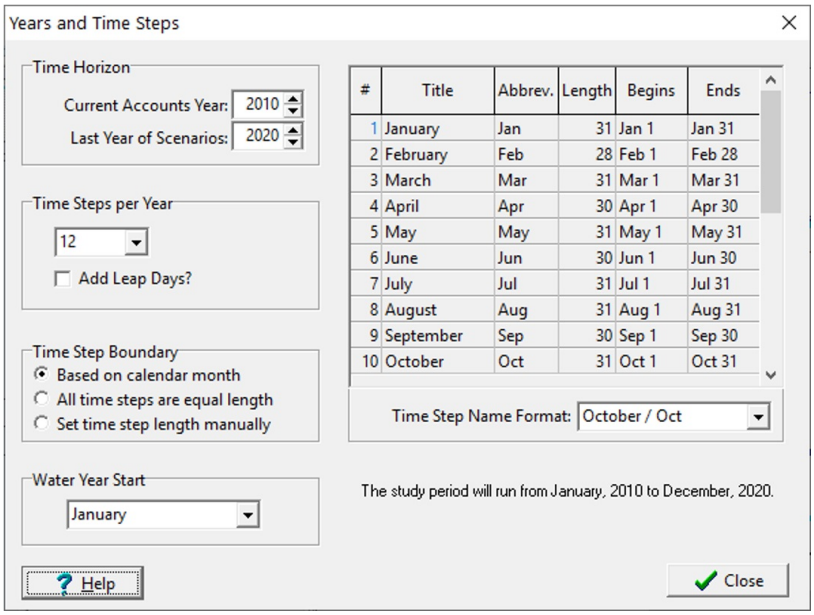


Figure 2.4: General Configuration in the WEAP System.

As a tightly integrated modeling, simulation, and analysis tool, the WEAP system consists of five views/parts to model different aspects of a water system. The structure of the model must be defined in the Schematic view. A water model in the WEAP system is a graph of node and link entities shown in the Schematic view (the solid red box in Figure 2.5a). The predefined node and link entities can be used to construct the nodes and links assigned to a geospatial map (the dotted blue box in Figure 2.5a). Table 2.1 lists the definition and usage of the WEAP's entities. The main entities are *River*, *Diversion*, *Reservoir*, *Groundwater*, *Desalination Plant*, *Demand site*, *Catchment*, *Wastewater Treatment Plant*, *Runoff*, *Transmission Link*, *Return Flow*, *Run of River Hydro*, *Flow Requirement*, and *Streamflow Gauge*. Some entities are presented as nodes (e.g., *Demand Site* and *Groundwater*), and some are presented as links between two nodes (e.g., *Transmission* and *Return flow*). The preliminary information about a node (e.g., the name property) must be set in the

Schematic view. Also, status (active or deactivate) and priority can be set for nodes and links. In Figure 2.5a, the number between parenthesis next to each entity type (e.g., *River* and *Demand Site*) shows its quantity. The number between parenthesis for each entity in the geospatial map indicates its priority relative to other entities during simulation executions.

Table 2.1: The Definition and Usage of the WEAP System Entities.

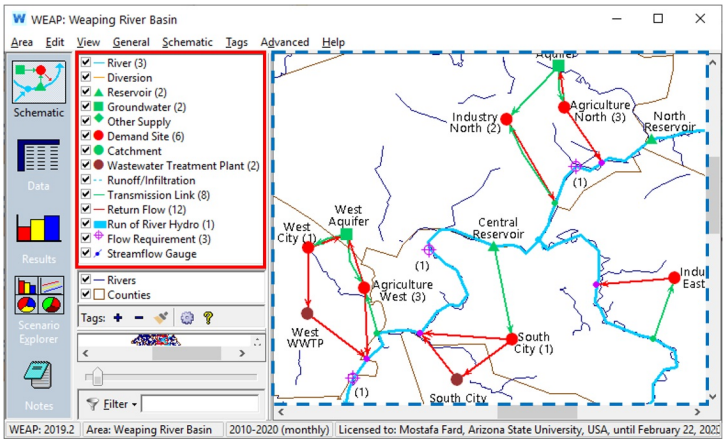
Entity	**Definition / Usage**
River	A collection of River Nodes to flow in or flow out the water in the system.
Diversion	Same as River but cannot flow out the water.
Groundwater	Represent the groundwater resource in a water system.
Other Supplies	To show a water supply but without storage capability between months.
Demand Site	A set of water users that share a physical distribution system.
Catchment	Specify processes such as precipitation, evapotranspiration, snow, and ice accumulation, melt, runoff, irrigation, and yields on agricultural and non-agricultural land.
Wastewater Treatment Plant	Represent the water treatment process in the system. Receiving water from demand sites, removing pollutants, and then returning treated effluent.
Runoff	Represent the water due to snow and ice melt, irrigation, and soil moisture storage that is not consumed by evapotranspiration or losses to increased soil moisture.

Continuation of Table 2.1	
Transmission Link	Deliver water from surface water (reservoir and withdrawal nodes), groundwater, and other supplies to satisfy final demand at demand sites.
Return Flow	Unconsumed water at a demand site can be directed to one or more demand sites, wastewater treatment plants, surface, or groundwater nodes via a Return Flow.
River Nodes	**- Reservoir** nodes represent reservoir sites on a river. **- Run-of-River Hydropower** nodes define points on which run-of-river hydropower stations are located. **- Flow Requirement** nodes define the minimum instream flow required at a point on a river or diversion to meet water quality, fish & wildlife, navigation, recreation, downstream or other requirements. **- Withdrawal** nodes represent points where any number of demand sites receive water directly from a river. **- Diversion** nodes divert water from a river or other diversions into a canal or pipeline called a diversion. **- Tributary** nodes define points where one river joins another. **- Return Flow** nodes represent return flows from demand sites and wastewater treatment plants. **- Streamflow Gauges** represent points where actual streamflow measurements have been acquired and can be used as points of comparison to simulate flows in the river.

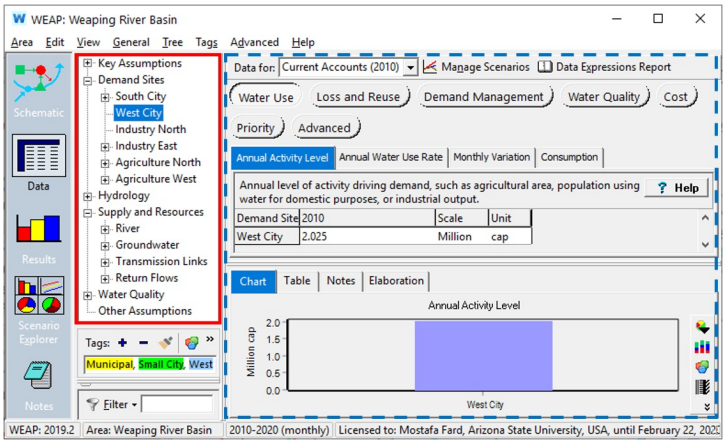
The defined entities in a Schematic view have predefined inputs and outputs (called data and result variables in the WEAP system). The next part in the WEAP system is the Data view which serves to parameterize the inputs and equations for the entities in a tree structure. The tree has "Supply and Resources", "Demand Sites and Catchments", "Hydrology", "Water Quality", and "Key Assumptions" categories (the solid red box in Figure 2.5b)). The Data view allows a modeler to create variables and relationships, enter assumptions and projections using mathematical expressions, the time-series wizard, and link to external files (e.g., CSV or Excel data files). The input variables of an entity are separated into different categories according to their usage. For example, Figure 2.5b (the dotted blue box) shows the West City demand site has seven categories ("Water Use", "Loss and Reuse", "Demand Management", "Water Quality", "Cost", "Priority", and "Advanced"), and "Annual Activity Level", "Annual Water Use Rate", "Monthly Variation", and "Consumption" variables belong to the "Water Use" category.

The Results view is for choosing the outputs of the simulation to be extracted and viewed in charts, tables, and on the Schematic view (see Figure 2.5c). Also, different entities, scenarios, years, and units can be used as plots displaying variable values for time-steps. The data can be filtered for a detailed and flexible display of the model input and output data values for time-step trajectories. The Scenario Explorer view can be used to select experiments to be observed and stored for post-processing. It provides the facility to observe the changes in the selected outputs by changing inputs. The Notes view provides a place to add the documentation for each entity.

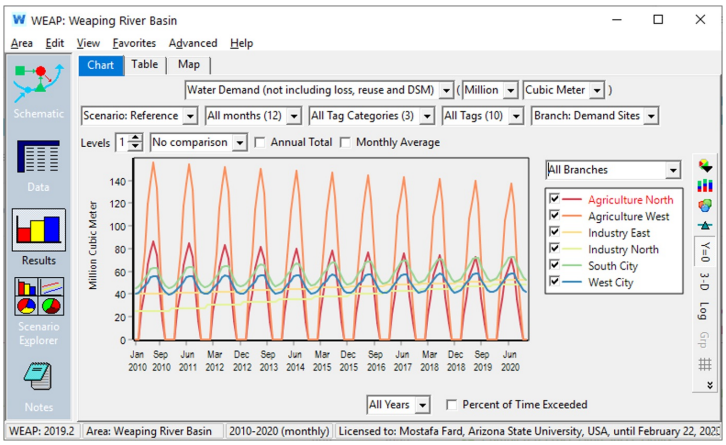
From the modeling perspective, the structure of the model (entities and their connections) must be defined in the Schematic view. Then, the input data will be injected into the input variables of the model via the Data view. Finally, after the simulation execution, the output data of the model is observable via the Results view.



(a)



(b)



(c)

Figure 2.5: The WEAP System Views. (a) Schematic View (b) Data View (c) Results View

The experiments for a model in the WEAP system are designed via scenarios and general configuration (see Figure 2.4). The Scenario Explorer and Notes views of the WEAP system do not impact the structure or behavior of a model.

From the simulation perspective, the WEAP system is based on Discrete-Time System Specification (Zeigler et al., 2018), and it has an uninterruptible execution. All input data must be ready before the start of every simulation. According to the general configuration, all output data will be accessible after a whole simulation execution period (from the start year to the end year). Interrupting the WEAP simulation midstream and resuming it is not allowed. In this respect, WEAP models are not reactive since they cannot have input from any external simulation model while being executed.

The WEAP system has predefined entities for common supply resources such as reservoirs, transmission links, and demand sites. The WEAP development team can add new entities due to the WEAP system being proprietary. Even though entities with their input and output variables are known, their mass-balance equations cannot be discovered from outside. The variables for the entities appear to be shared within the WEAP system. A water model's logical and schematic parts are tightly interwoven into the scenarios, general configuration, and results.

The WEAP system uses mixed-integer linear programming (MILP) to optimize the satisfaction of requirements for the demand sites, reservoir filling, user-specified instream flows, and hydropower entities subject to demand priorities, supply preferences, mass balance and other constraints (SEI, 2022d). The WEAP system supports LPSolve, XA, and Gurobi MILP solvers. The LPSolve is open source and included in the WEAP system. XA and Gurobi are commercial products (Gurobi, 2022). For very large models, the commercial solvers can perform faster. The simulation results can vary slightly depending on the selected solver.

2.4.2 Low Emission Analysis Platform System

The Low Emissions Analysis Platform (LEAP) is an integrated modeling tool that can be used to track energy consumption, production, and resource extraction in all sectors of an economy. The LEAP was initially created in 1980 on a mainframe computer. In 1983, with funding from US-AID, it was converted for use on a minicomputer, and the first user interface was added. By 1985, LEAP had been ported again to the newly emerging IBM PC microcomputer, making wider dissemination and a more user-friendly interface possible. In 1992, the first global energy study using LEAP was published (Lazarus et al., 1993). With climate change rising on the international agenda, LEAP has further enhanced as a tool for Greenhouse Gas (GHG) mitigation assessments. By the late 1990s, with support from the Dutch Government (DGIS), a new Windows-based version of LEAP was created, allowing the original goal of a highly user-friendly energy and environment planning tool. The first version of the new tool was made public in early 2001, which caused a significant increase in the number of LEAP users (see Figure 5). In 2012, the LEAP system was linked to the WEAP system to support Water-Energy Nexus (WEN) modeling analyses. Domain experts and researchers widely use the WEAP and LEAP tools for WEN analyses (Howells et al., 2013).

The LEAP system can be used to account for both the energy sector and non-energy sector greenhouse gas (GHG) emission sources and sinks. In addition to tracking GHGs, LEAP can also analyze emissions of local and regional air pollutants (SEI, 2022b). LEAP is intended as a medium to long-term modeling tool. Most of its calculations occur on an annual time-step, and the time horizon can extend for an unlimited number of years. Studies typically include both a historical period known as the Current Accounts, in which the model is run to test its ability to replicate

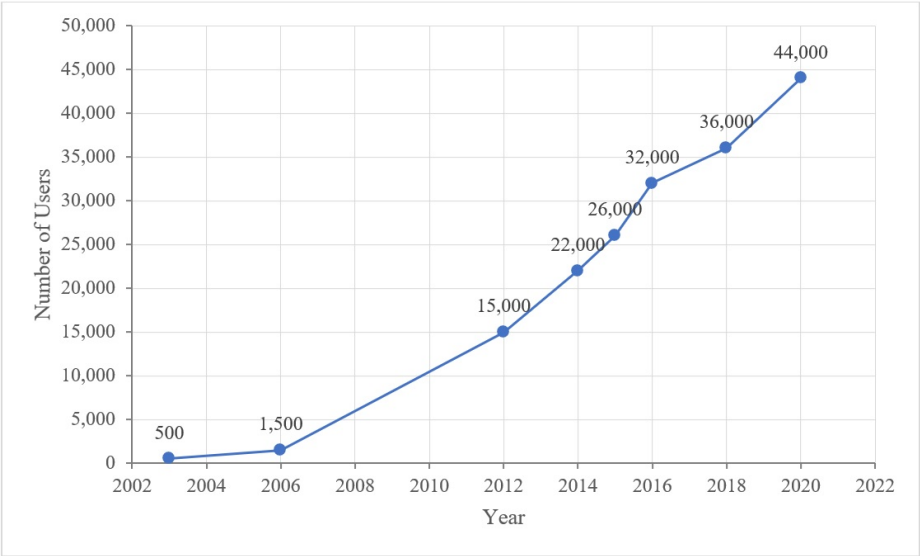


Figure 2.6: Number of LEAP's Users Since 2003.

known statistical data, as well as multiple forward-looking scenarios. Some results are calculated with a finer level of temporal detail. For example, for electric sector calculations, the year can be split into different user-defined "time-slices" to represent seasons, types of days, or even representative times of the day. These slices can be used to examine how loads vary within the year and how electric power plants are dispatched differently in different seasons.

LEAP is designed around the concept of long-range scenario analysis. Scenarios are self-consistent storylines of how an energy system might evolve. Using LEAP, policy analysts can create and then evaluate alternative scenarios by comparing energy requirements, social costs and benefits, and environmental impacts. The LEAP Scenario Manager can be used to describe individual policy measures, which can then be combined in different combinations and permutations into alternative integrated scenarios. This approach allows policymakers to assess the marginal impact of an individual policy and the interactions that occur when multiple policies and measures are combined. For example, the benefits of appliance efficiency standards combined

with a renewable portfolio standard might be less than the sum of the benefits of the two measures considered separately.

As a tightly integrated modeling, simulation, and analysis tool, the LEAP tool consists of seven views/parts. In the Analysis view, the structure of the model must be defined using five types of entities; *Demand*, *Transformation*, *Process*, *Resource*, and *Effect* (the solid red box in Figure 2.7a). Table 2.2 lists the definition and usage of the LEAP's entities. The defined entities have predefined inputs and outputs (called data and result variables in the LEAP system). This view also serves to parameterize the inputs and equations for the entities in a tree structure with the Demand, Transformation, Resource, and Key Assumptions categories (the dotted blue box in Figure 2.7a). The Analysis view allows a modeler to create variables and relationships, enter assumptions and projections using mathematical expressions, the time-series wizard, and link to external files (e.g., CSV or Excel data file). The input variables of an entity are listed according to the selected entity or branch. For example, Figure 2.7a (the dotted blue box) shows the Mark Wilmer demand has two variables; Activity Level and Final Energy Intensity Time Sliced.

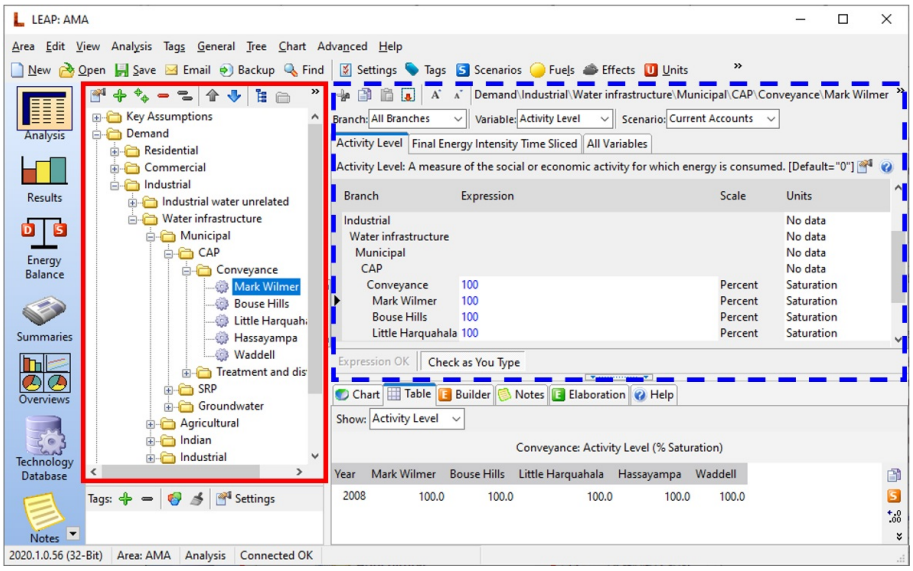
The Results view displays results in charts and tables for all parts of the energy system (see Figure 2.7b). Also, different entities, scenarios, years, and units can be used as plots displaying variable values for time-slices. The Energy Balance view can be used to select experiments to be observed in a standard energy balance table, chart, or Sankey Diagram. The Summaries and Overviews views are used to create customized tabular reports and store data for post-processing. They provide the facility to observe the changes in the selected outputs by changing the inputs. The Technology and Environmental Database (TED) view provides extensive information describing the technical characteristics, costs, and environmental impacts of energy technologies available internationally or in particular developing country regions. The

Table 2.2: The Definition and Usage of the LEAP System Entities.

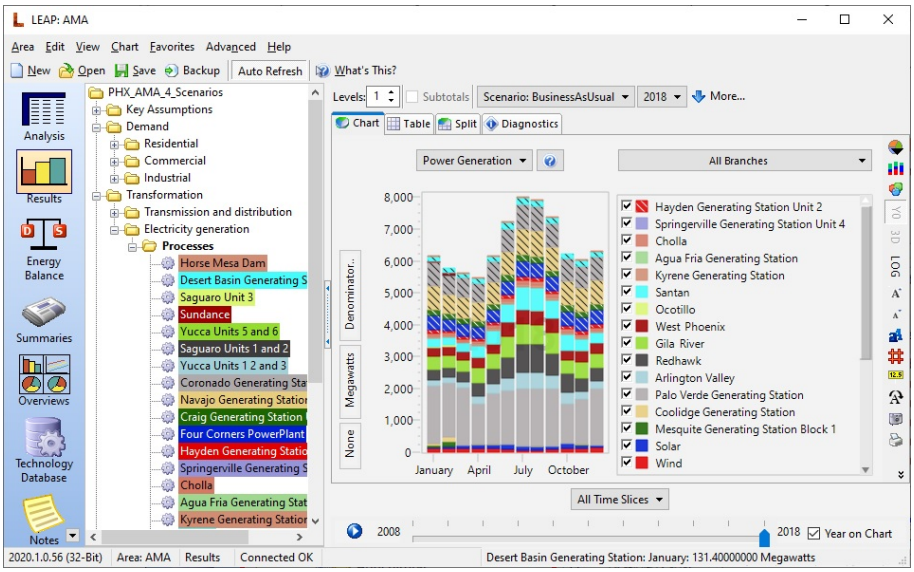
Entity	Definition / Usage
Demand	A dis-aggregated end-use based approach for modeling the requirements for final energy consumption. The energy demand analysis methodology can be Activity Level Analysis, Stock Analysis, or Transport Analysis.
Transformation	Simulate of the conversion and transportation of energy forms from the point of extraction of primary resources and imported fuels to the point of final fuel consumption.
Process	Represent the unique technologies that convert energy from one form to another or transmit or distribute energy, such as individual power plants or groups of power plants.
Resource	Represent the availability of primary resources (e.g., fossil, wind, solar, nuclear, etc.)
Effect	Represent the environmental effect (e.g., air pollutant, soil pollutant, etc.) produced in the energy network.

Notes view provides a place to add the documentation for each entity. From the modeling perspective, the structure of the model (entities and their connections) and the initialization for the inputs must be defined in the Analysis view. Then, the output data of the model is observable after the simulation execution via the Results view. From the simulation perspective, the LEAP system (like the WEAP system) is based on Discrete-Time System Specification, and it has an uninterruptible execution.

The LEAP system uses two frameworks for optimization calculation for Linear Programming (LP) and Mixed Integer Linear Programming (MILP); The Open Source Energy Modeling System (OSeMOSYS) and the Next Energy Modeling sys-



(a)



(b)

Figure 2.7: The LEAP System Views. (a) Data View (b) Results View

tem for Optimization (NEMO) (SEI, 2022c). Table 2.3 provides a quick comparison of the OSeMOSYS and NEMO frameworks (Institute, 2022). The GLPK and CBC solvers are free, and CPLEX, Gurobi, MOSEK, and XPress are commercial.

Table 2.3: Comparison Between Osemosys and Nemo Optimization Frameworks.

Feature	**OSeMOSYS**	**NEMO**
Developer	KTH	SEI
Installation	Integrated into LEAP	Via Separate Download
Platform	GLPK (last updated 2018)	Julia (actively developed at MIT)
Open Source	Yes	Yes
Licensing	Free & Include with LEAP	Free. It can be downloaded from the LEAP website (no separate license required)
Small Data Set	Faster	Fast
Large Data Set	Slow	Fast
Time Slicing	Limited Flexibility	Very Flexible
Energy Storage	No	Yes
Solvers	GLPK, CPLEX	GLPK, CPLEX, CBC, Gurobi, MOSEK, XPress
Parallel Developed	Only when using CPLEX	Yes
Active Status	Unknown	Yes, by SEI
Network and Power Flow Simulation	No	Yes, in NEMO & Coming to LEAP/NEMO

2.4.3 WEAP & LEAP Data Schema

Figure 2.8a presents the data schema related to an entity from the outside perspective for a prototypical model and scenario. The data schema in Figure 2.8a has

three axes – the y-axis is used for variable names, the z-axis is used for the years of a simulation, and the x-axis is used for yearly time granularity. Each entity (e.g., a demand site in the WEAP system or Transformation in the LEAP system) has several input and output variables. A variable can have annual time granularity like variable v_2 shown in Figure 2.8a which has one value per year or finer time granularity (for example, monthly, weekly, daily, and so on) like variables v_1 shown in Figure 2.8a. The time granularity for variables that do not have annual time-steps must be defined in the General Configuration (Time Step per Year section in Figure 2.4 and the same configuration in the LEAP system). Each cell in Figure 2.8 has a float-type data value according to its defined time granularity. Years and time-steps/time-slices in each year have ascending order. For example, the green (which has a circle sign) and orange (which has a cross sign) cells in Figure 2.8a present the first and last timestamp's values for the variable v_1 in a simulation scenario. Considering Figure 2.8a, there are $m \times q$ timestamp's values (number of years $\times$ number of time-steps/time-slices per year) for variable v_1. The variable v_2 for the same simulation experiment has m timestamp's values for years y_1, y_2,..., y_m, each having one time-step/time-slice.

Figure 2.8b shows the data schema for one variable related to multiple scenarios. Considering any variable in Figure 2.8a, the y-axis is used for the years of a simulation, the z-axis is used for the number of simulation scenarios, and the x-axis is used for yearly time granularity. For example, suppose variable v_1 is selected from Figure 2.8a and expanded for multiple scenarios. The result will be Figure 2.8b, which presents the data cell values for scenarios s_1, $\cdots$, s_r for years y_1, ..., y_m which each cell is divided to q equal time-steps/time-slices. It is important to note that the values in the lowest horizontal level (the yellow cells) are the same for all scenarios because they represent the values for the Current Accounts scenario. Other scenarios affect the values of a variable from the start year (start year + 1) to the end year.

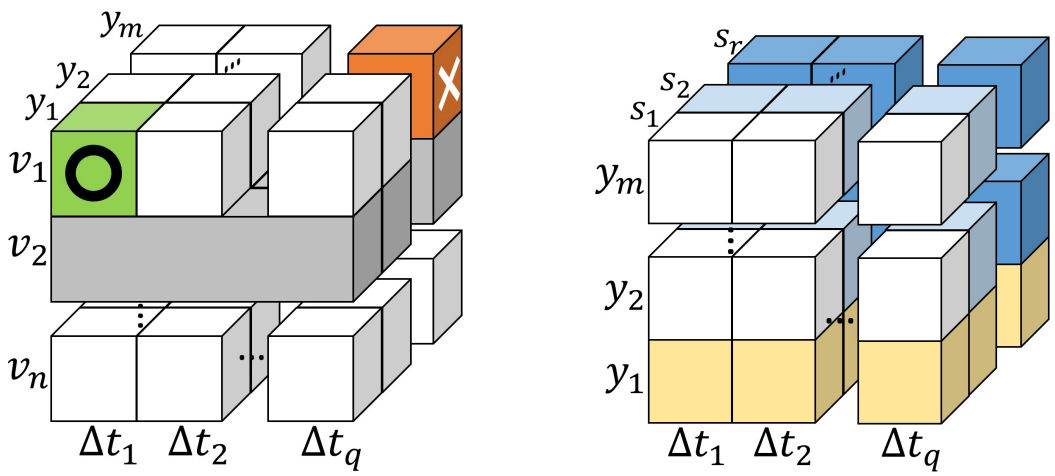


Figure 2.8: WEAP/LEAP's Component Data Schema (a) Data of Different Variable given a Scenario. (b) Data of a Variable given Different Scenario.

Both WEAP and LEAP systems support VB-Script, JavaScript, Perl, and Python languages using the standard COM Automation Server to access the entities, variables, data, and execution control. Even though the entities with their input and output variables are known, their mass-balance equations are unknown and cannot be discovered from outside. The variables for the entities are shared within the WEAP and LEAP tools. A water/energy model's logical and schematic parts are tightly interwoven with the scenarios, configurations, and results.

Chapter 3

LITERATURE REVIEW

3.1 Water and Energy Modeling

There exist a wide variety of many tools for modeling and simulating water and energy systems, serving purposes ranging from natural processes to engineered distribution networks. Such software tools are developed by representing water and energy systems as data sets with functions, objects, and services. Legacy and object-oriented software systems can be encapsulated as services in Service Oriented Architecture (SOA) paradigm. Various approaches have been proposed for transforming legacy software systems to be integrable with other software systems (Bisbal et al., 1999). One of these approaches is "wrapping" where any proprietary legacy software system with input/output API (e.g., WEAP and LEAP systems) can be encapsulated inside other software systems (Sneed, 2006; Sneed et al., 2006). Individual functions in the legacy software are wrapped into web services. New components are designed according to the code segments that perform a service or data modification. Each new component is given a Web Services Description Language (WSDL) interface and targeted for either SOAP or RESTful framework. This research follows the rationale and the general approach of transforming a legacy system to flexible service-oriented software frameworks in addition to component-based modeling and simulation (Zeigler et al., 2017).

Services and service compositions with resource management supporting load balancing and context storage are proposed for legacy modeling and simulation tools (Pullen et al., 2005). This approach is applied to a legacy solid multibody simula-

tion. This tool consists of five Fortran programs which are wrapped individually in Java-based web-services. The Message Passing Interface (MPI, 2022) concept is used to define the communication between process (or communication) logic and domain logic with workflows/service compositions. The result improves the existing legacy application and speeds up the overall simulation execution by automating manual tasks and parallelizing web-services to control, orchestrate, and visualize simulation experiments. The Componentized WEAP RESTful framework shares the wrapping legacy software applications; however, its primary goal is to support structured integration with other modeling and simulation tools and generally scientific applications.

An objected-oriented modeling framework without using the SOA paradigm is proposed for simulating watershed flow and sediment processes at the catchment scale based on fine-grained components (Qin et al., 2019). This work is based on the systems-of-systems concept to build new water-related models quickly and effectively by de/composing models in the manner of plug and play of user-defined components. The objects from decomposition are encapsulated into corresponding components using dependency injection, a technique for achieving separation of concerns, to define the relationships and coupling of different components.

A web-based platform has been introduced to support efficient multivariate visualization of environmental data from sensor observation networks (Li et al., 2016). Data are collected based on the SOAP framework and XML format from different locations/resources for creating various visualizations and analyses using JSON. This platform uses a caching system (collected from distributed sensors via web-services) to store data in databases (PostgreSQL for locally storing and Hadoop-based OpenTSDB for distributed storing) to increase data access efficiency. Also, a data cube model is established to reshape heterogeneous data and support unified data operations.

Agent-based modeling is also proposed and used for water resource management systems. Agents have their own goals and behaviors and can adapt and modify their behaviors (Akhbari, 2012). In water resources systems modeling, agents can be individual ground and surface water users, water polluters, various infrastructural elements, cities, or policymakers on different levels (Nikolic & Simonovic, 2015). A framework to model and simulate water supply and demand for urban households has been developed based on Agent-Based Modeling and System Dynamic modeling (Alvi et al., 2018). This is an approach to model water management at micro (short-term) and macro (long-term) abstraction levels. Another approach for simulating urban water resource management uses a multiagent Q-learning-based allocation agent-based algorithm (with adaptive reward value function to improve the performance) (Ni et al., 2013). This algorithm supports allocating water resources efficiently among stakeholders. An agent-based framework has also been developed to simulate the behavioral characteristics of urban water users while accounting for their social interactions (Darbandsari et al., 2017). A model has interactions between agents as well as agents and environments. The focus of this framework is to help study and evaluate the impact of different climate and government policies. Although agent-based frameworks are inherently grounded in the concepts of components, they are not as flexible and scalable as service-oriented frameworks and thus can be challenging to be loosely integrated with other tools.

The WEAP system is used as the primary tool for modeling and simulating water management under different socio-economic and climate scenarios for regions including South Africa, China, Greece, Benin, and Pakistan (Amin et al., 2018; J. Gao et al., 2017; Höllermann et al., 2010; Lévite et al., 2003; Psomas et al., 2016). This suggests that the WEAP system's componentization can help combine these and other WEAP water simulations with separate simulations for agriculture, climate, and energy sys-

tems. Given the related works highlighted above, the Componentized WEAP is a tool built using the RESTful SOA architecture, and the data cube structure allows integrating it with tools for simulating energy and food systems (Fard & Sarjoughian, 2020, 2021b).

3.2 Tools/Frameworks to Model the FEW-Nexus

Software frameworks for combining models can be categorized into a *Common Library*, *Product Line Architecture*, *Interoperability Protocol*, *Object Model*, *Formal*, and *Integrative Environment* (Petty et al., 2014). The *Common Library* simulation framework is based on a collection/set of software modules (without standalone execution) reusable through conformance to standard interfaces that allow the modules to interoperate with the other modules. The *Product Line Architecture* simulation framework is based on the planned development of multiple related simulation products that, to the extent possible, share common software components. The *Interoperability Protocol* simulation framework is based on the run-time exchange of simulation data or services. The *Object Model* simulation framework is based on a standard for component specifications (not implementation). Typically, the components are not themselves standalone simulation systems but rather compose each other in the context of an overall simulation system. The *Formal* simulation framework depends upon a mathematical notation to define the components (usually models), compositions of models, and the interfaces between them. Finally, the *Integrative Environment* simulation framework is an execution environment used to connect components that may have been written with no initial intent to interoperate. Usually, specialized software wrappers and scripts are part of the simulation framework to connect the simulation components. From this vantage point, our research can be placed in the *Interoperability Protocol* software frameworks.

Numerous and diverse tools/frameworks have been presented in different contexts since the interpretation and centrality of the FEW-Nexus. Those tools/frameworks can be roughly categorized into resource-environmental footprint quantification, assessment and systematic simulation, and optimal management / integrated models (P. Zhang et al., 2019). The resource-environmental footprint quantification models are widely used to quantify the resource and economic efficiency associated with the FEW system, which is consistent with the "interdependence" property of the nexus concept (e.g., UWOT (Baki & Makropoulos, 2014) and REWSS (A. T. Dale & Bilec, 2014)). The assessment and systematic simulation models help to assess and model the performance of the FEW system (e.g., WEF Nexus Tool 2 (Daher & Mohtar, 2015) and CMDP (Nanduri & Saavedra-Antolnez, 2013)). Three types of methods have been used in this domain; indicator systems, system dynamics, and network analysis models. The optimal management/integrated models consider multiple systems and the interactions between them as an intricate process involving several modeling and computational complexities (e.g., CLEWS (Howells et al., 2013) and WEAP-LEAP (SEI, 2022b; Sieber et al., 2005)). The last model has more complexity (based on the aggregation of qualitative and quantitative measures) than the first models (Dargin et al., 2019). Our research is in the category of integrated models.

In recent years, there has been a significant increase in studying the Water-Energy Nexus (WEN). The number of studies and the ability of the scientific community to assess has been on the rise, mainly to understand the water and energy interactions. One study examined 35 methods, tools, and frameworks related to the WEN based on the geographical scale and the nexus scope, focusing on the interactions between water, energy, and others, including environment, food, land, and climate (Dai et al., 2018). The Stockholm Environment Institute conducted research on integrated analysis of water, energy, and greenhouse gas (GHG) emissions. For example, a study

on the water and energy systems in Sacramento, California, over the period 1980–2001 with weekly time-step was undertaken on four climate scenarios that represent the impact of future temperature and precipitation extremes (L. L. Dale et al., 2015). In another study, four possible scenarios with a direct impact on demand and supply water and energy systems on the Jajrood river, Tehran, Iran, over the period 2016-2026 with monthly time-step was undertaken (Javadifard et al., 2020). The impact of changing water demand, GHG emissions, and cost-effectiveness via nine different scenarios on the Western Canadian province of Alberta is another study (Agrawal et al., 2018). This study forecasts water consumption and GHG emissions from the power sector for the 2015-2050 period.

Considering simulation studies of integrated food, energy, and water systems, frameworks, and tools such as Precipitation Runoff Modeling System (PRMS) (Markstrom et al., 2015) and Ground-water and Surface-water FLOW (GSFLOW) (Markstrom et al., 2008) and WEF-Nexus Tool 2.0 (Daher & Mohtar, 2015) have been developed. These tools are not based on component-based modeling principles and service-oriented computing. The PRMS is a deterministic, distributed-parameter, physical process-based modeling system developed to evaluate the impacts of various combinations of climate and land use on surface-water runoff, sediment yields, and general basin hydrology. The GSFLOW is designed to simulate coupled ground-water and surface-water systems. The WEF Nexus Tool 2.0 is a scenario-based tool for guiding resource allocation at the country level for a given level of food self-sufficiency and a set of technologies, land uses, and resource availabilities. These approaches and tools, unlike those briefly described above, can provide limited capabilities needed for integrating food, energy, and water models developed in different simulation tools (Sarjoughian, 2006).

The Climate, Land, Energy, Water (CLEW) framework is likely the most relevant work for our research. The CLEW framework is based on a system's approach to analyzing interactions between interconnected sectors (Kaddoura & El Khatib, 2017). It uses existing simulation tools (WEAP, LEAP, and AEZ) based on a modular structure to illustrate synergies and trade-offs within the CLEW areas for decision-making related to achieving development goals (Howells et al., 2013). The CLEW framework applies to different geographical scales from global to regional, national (e.g., Mauritius, Uganda and Nicaragua, Bolivia, and Kenya), and urban levels (e.g., Oskarshamn and New York City) ("CLEWS-Home", 2022).

The Nexus Simulation System (NexSym) models the local techno-ecological interactions relevant to the FEW-Nexus by integrating models for ecological, technological, and consumption components. It allows the user to build, simulate, and analyze a "flowsheet" of a local system (Martinez-Hernandez et al., 2017). Users must define the system level and simulation parameters, then use the component-flow diagram to draw the virtual local production system, finally execute the tool to process the model, and generate the results (in tables and plots). The NexSym allows exploring not only how parts of the nexus are affected by a change in another part but also to evaluate key interactions that could be developed into synergistic integrations (Martinez-Hernandez et al., 2017).

The integrated WEAP and LEAP systems use a hidden internal linking mechanism to interact with one another. In each system, a table specifies the mappings between the WEAP and LEAP scenarios and time granularities for given projects. Two restrictions must be satisfied for this connectivity. First, both projects must have the same start and end years for a simulation. Second, both projects must have a matching set of time-steps (e.g., monthly and daily); otherwise, some data will be lost. After establishing the connection, a system has access (via the equations/ex-

pressions) to read the Data and Result variables of different entities from the other system.

The WEF Nexus Tool 2.0 is a web-based tool for guiding resource allocation at the country level, for a given level of food self-sufficiency, and a set of technologies, land uses, and resource availabilities (Daher & Mohtar, 2015). The modeling steps in the tool start by defining the scenarios, then quantifying the flows of matter and energy between the three nexus areas. The tool provides seven outputs (total water requirement, total land requirement, local energy requirement, local carbon footprint, financial cost, energy consumed through import, and carbon emissions through import) based on the user-defined scenarios to calculate an overall sustainability index (for scenario comparison). Unlike the CLEW framework, the WEF Nexus Tool 2.0 allows the user to factor economic modules into the scenarios (Kaddoura & El Khatib, 2017).

The Multi-Scale Integrated Analysis of Societal and Ecosystem Metabolism (MuSIASEM) is an open framework that aids in determining the feasibility and desirability of socioeconomic systems. It uses Complex System Theory concepts (multi-scale accounting, multi-purpose grammar, impredicative loop analysis) and Bioeconomic concepts (flow-fund model) to simultaneously assess technical, economic, social, demographic, and ecological variables (to normalize and analyze data from different hierarchal scales, such as national and sub-national levels). The MuSIASEM framework can be used for diagnostics (a snapshot of a society's current metabolic process) or simulations (analysis of possible scenarios from the perspectives of feasibility, viability, and desirability). Furthermore, forecasting is not possible using MuSIASEM because it does not calculate benefits and costs neither provide typical technical variable outputs over time.

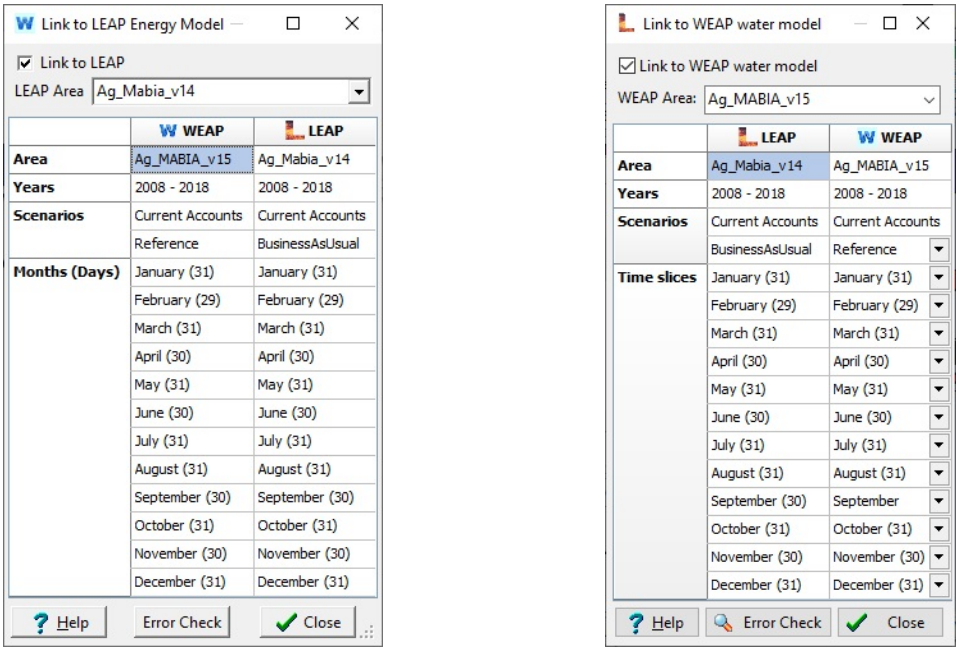
According to the research done by Dargin et al. on the FEW tool's complexities (significant data requirements and resource intensity), tools receiving higher complexity scores (e.g., CLEW, MuSIASEM, WEAP, and LEAP), while being able to capture details to specific resource interactions, are unable to cover a larger number of interactions and system components simultaneously, as compared to lower complexity score tools (e.g., WEF Nexus Tool 2.0) (Dargin et al., 2019). While many studies aim to develop new methods and frameworks to comprehensively assess different nexus scopes, none can or do provide a singular framework for performing the WEN nexus.

Two comprehensive studies focusing on water and energy for the Phoenix AMA were carried using the WEAP and LEAP systems. In one study, five possible future scenarios were developed and analyzed for the 2010-2060 period (Guan et al., 2020). In the other study, four possible forecasts of the future energy demand and supply are generated for the 2019-2060 period (Mounir et al., 2019). These research findings underscore the importance of modeling and understanding water and energy systems interactions flexibly and at high fidelity.

3.3 WEAP-LEAP Internal Linkage

From a model coupling perspective, From a model coupling perspective, the WEAP and LEAP systems have an internal linking mechanism established in 2012 that can bi-directionally share data to read variables from one to another. As shown in Figure 3.1, in each system, a table is displayed in a form to specify mappings between the WEAP and LEAP scenarios and time granularities for given projects. Two restrictions must be satisfied for this connectivity. First, both projects must have the same start and end years for a simulation (same time interval). Second, both projects must have a matching set of time-steps, e.g., monthly and daily (same time granularity). Otherwise, some data may be lost during the coupling. After establishing the

connection, a system has access (via the expressions) to read the Data and Result variables of different entities from the other system.



(a) (b)

Figure 3.1: WEAP/LEAP's Component Data Schema. (a) Internal Linkage Mapping in the WEAP System. (b) Internal Linkage Mapping in the LEAP System.

The WEAP and LEAP systems can bi-directionally share data with one another. Each system can read the Data and Result variables of different entities from the other system. The path to an entity and a specific variable must be defined via form wizards (in the tree view for the entities and the list view for the variables) or by writing the expression via text wizards. Paths are difficult to define using wizards when the user wants to manipulate more than one variable from another system for a variable in another system. Users must know the details of the models in the WEAP and LEAP systems. Using the wizards becomes increasingly more challenging as the scales of the water and energy models increase. The *"WEAPValue(entityPath:variableName[unitName])"* function needs to be defined in a LEAP model for accessing a variable in the WEAP model. And, the *"LEAP-*

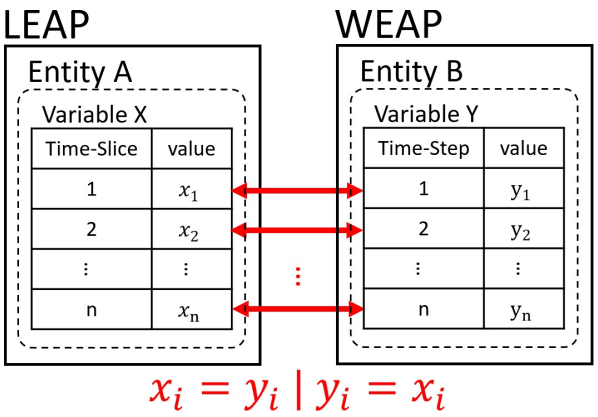
Value(entityPath:variableName[unitName])" function needs to be defined in a WEAP model in the same way for accessing a variable in the LEAP model. All algebraic equations, some of which can be complex, must be defined in the expression part of the variable of the destination model. Defining a variable in a water model to use variables from multiple entities in an energy model is cumbersome. Defining paths between an energy model with many variables that use many variables in a water model becomes complicated and error-prone.

The variables in the WEAP and LEAP models can be mapped to one another using the time resolution types I, II, and III shown in Figure 3.2. The arrow directions in the figures signify the data of a target time-step/time-slice is read from the data of source one. Also, it is supposed that the first time-step/time-slice is mapped to the first time-slice/time-step, the second time-step/time-slice to the second time-slice/time-step, and so on in the WEAP-LEAP Internal Linking form (see Figure 3.1). For the Type I time resolution, both variables have the same number of time-steps and time-slices in the WEAP and LEAP systems, respectively. Figure 3.2a shows that the time resolution for the water and energy models are the same. For the Type II time resolution, one of the variables (from the WEAP or LEAP) has a smaller time resolution, and the variable with the smaller time resolution reads the variable with a larger time resolution from the other system. In Figure 3.2b, the variable X in the LEAP model with yearly time-slice is read by the variable Y in the WEAP model with n time-steps. As a result, the x_1 variable is used by all time-steps for values y_1,y_2,. . .,y_n of variable Y. From a yearly perspective, the value x_1 which is for the whole year of the variable X in the LEAP system is multiplied by n for variable Y in the WEAP system. This problem can be solved by applying a distribution on a smaller time resolution. For example, by dividing the x_1 by n on the WEAP side in Figure 3.2b for uniform distribution. For the Type III time resolution, one

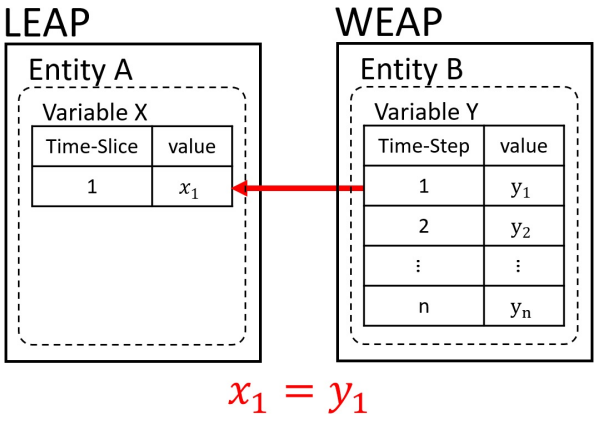
of the variables (from the WEAP or LEAP) has a smaller time resolution, and the variable with a smaller time resolution is read by the other variable. In Figure 3.2c, the variable Y in the WEAP system with n time-steps is read by the variable X in the LEAP system with yearly time-slice. The value of the first time-step of variable Y is read by the variable X and the remaining values y_i $(i = 2, \ldots, n)$ are ignored. This restriction cannot be removed using the WEAP-LEAP internal linking, and some data is lost. A well-defined specification is presented in the proposed coupled WEAP-LEAP framework for all types of interactions in the WEAP-LEAP internal linkage (based on the time resolutions).

Defining the time granularity for the water and energy models has an important distinction in the WEAP and LEAP systems. The time-step in the WEAP system divides a year into a finite equal number of segments, for example, yearly (one step per year), monthly (12 steps per year), or daily (365 steps per year). Unlike time-steps, the time-slices must be defined one by one by the user, and the size of the segments can be different; however, the aggregation of time-slices must cover the whole year. For example, the first time-slice can cover the first three months of a year, and the next time-slice can cover the rest (9 months).

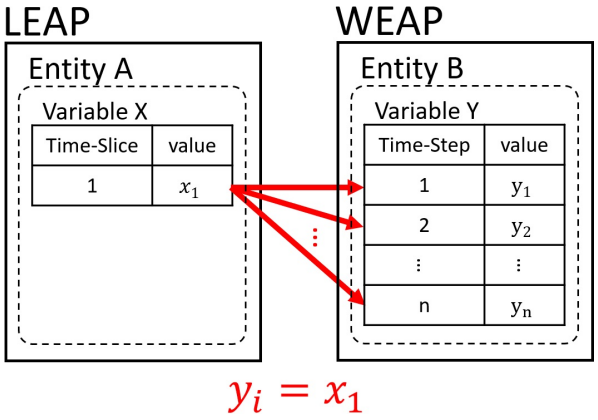
Domain experts and researchers widely use the coupled WEAP and LEAP tools for WEN analyses (Howells et al., 2013). The WEAP and LEAP models' scenarios and time granularities must be mapped for given projects. Some restrictions must be satisfied for this connectivity. First, both projects must have the same start and end year for the simulation. Second, both projects must have a matching set of time-steps, e.g., monthly or daily (see section 5 of (Fard & Sarjoughian, 2020) for more details). Third, WEAP and LEAP systems must be installed on the same machine. Forth, the WEAP and LEAP tools must be executed manually and sequentially. After coupling the WEAP and LEAP systems, they can bi-directionally share data. Nevertheless,



(a)



(b)



(c)

Figure 3.2: The Interaction Types in the WEAP-LEAP Internal Linking, Based on the Time Granularity. (a) Type I: Same Time Resolution for WEAP and LEAP Models. (b) Type II: Larger to Smaller Time Resolution for WEAP and LEAP Models. (c) Type III: Smaller to Larger Time Resolution for WEAP and LEAP Models.

users must know all details of the defined water and energy models in the WEAP and LEAP systems. Any changes in one system need strict consideration in the other system. In the WEAP-LEAP Internal Linking, the modeler must decide and manually execute data exchanges between the water and energy models. After the WEAP-LEAP Internal Linking, the execution of each system depends on the execution of the other one because a system may generate some data (as the output) which are used by the other system. All the input data for the whole simulation interval (from the start year to the end year) must be ready before simulation execution in the WEAP or LEAP systems. Also, all output results are generated after the simulation execution (before the execution, they are zero).

Consider a situation where each model reads the output of the other model as its input. Initially, both models have some data as their inputs and zero values as their outputs. Then, the modeler must manually execute either the water or energy model first. Consequently, the first model reads zero values (outputs) from the second model and generates its own results. Then, the second model is executed and read the outputs of the other system (which can be non-zero values). In this proposed coupled WEAP-LEAP framework, the order of data transformations is defined in the interaction model and supported with automated execution.

3.4 Developed Frameworks Based on the KIB Approach

Regarding the realization of the KIB approach, an interaction model is presented for composition between the DEVS and Composable Cellular Automata (CCA) models at formalism, system architecture, and model execution levels (Mayer, 2009). The interaction model is exemplified using an agent-environment hybrid model using the DEVS formalism to model the agents and the Geographical Resources Analysis Support System (GRASS) to model the environment. Another example of this interaction

model is implemented for modeling long-term dynamics of complex socio-ecological systems using stochastic land-use modeling coupled with the landscape evolution model (for land use, water, and landscape) (Barton et al., 2010). Continuing this context, a discrete-event composable cellular automata (CCA-DEVS) framework is proposed to compose discrete-time CCA and parallel DEVS models at the I/O level (C. Zhang et al., 2020). The DEVS-Suite simulator supports the modeling and simulation of CCA-DEVS models with complementary run-time textual and 2D displays (i.e., time-based input, state, and output trajectories). This approach can help better understand cell-to-cell and CCA-to-CCA interactions to develop alternative or better designs for compositions of multiple cellular automata. The interactions between CCAs in the CCA-DEVS framework can be modulated in terms of data transformations, control schemes, distributed execution, and spatiotemporal granularity.

In another research, the Geographic Knowledge Interchange Broker (GeoKIB) is proposed as a mediator to regulate unidirectional interactions between composed geographical models (Boyd & Sarjoughian, 2020). The GeoKIB performs the basic operations required of interaction models while providing functionality that facilitates spatial operations. Different input and output data types are supported using passive or active data transmission. Synchronization of time-tagged input and output values is possible via connections to shared simulation clocks. Using the GeoKIB, a spatial conversion algorithm can transform a two-dimensional geographic data map to another region (probably with different map cell sizes and boundaries). Indeed, the GeoKIB was designed to compose simulations that use the GRASS GIS system. A composition of a cellular automaton model (to represent changes to a geographic area over time) and an agent-based model are developed to demonstrate the functionality of the proposed approach. The interaction model presented in this article is based

on a formal specification. In contrast, the GeoKIB is grounded on object-oriented principles.

A framework using the KIB approach was developed for composing distinct models of the DEVS, Model Predictive Control (MPC), and Linear Programming (LP) models in the semiconductor manufacturing supply-chain systems (Huang et al., 2009). This framework provides a set of suitable message mappings and transformations. A causal parallel execution protocol with logical time synchronization was devised and used to develop a prototype distributed simulation framework for DEVSJAVA, MATLAB, and OPLStudio (linear optimization tool) tools. The interaction model can be defined using an XML file (unlike the design in this article which models can be defined using REST APIs and programming languages). Another example of implementing the KIB approach for supply-chain systems is the Optimization, Simulation, and Forecast (OSF) platform (Smith, 2012). The optimization and simulation models are developed in the OPLStudio/CPLEX (an optimization engine to develop LP models managed by IBM) and the DEVS-Suite simulator. The OSF also improved the structure of the XML file to create a multi-echelon model for better usability and scalability.

3.5 Interoperability

The High-Level Architecture (HLA) represents a major advance in Modeling & Simulation methodology as a standard mandated by the U.S. Department of Defense (DoD) to promote simulation interoperability and reuse (Dahmann et al., 1997). It provides the specification of a common technical architecture for reuse and interoperation (interoperability) across simulations. The baseline definition of the HLA includes the Rules, the Interface Specification, and the Object Model Template. An HLA-based architecture must have modular components with well-defined function-

ality and interfaces. Further, it must separate the functionality needed for individual simulations from the infrastructure required for interoperability among simulations (Dahmann et al., 1997).

There is a fundamental distinction between the Composability of Models and Interoperability of Simulations viewed from the theory of modeling and simulation on the one hand and HLA standardization on the other hand (Sarjoughian & Zeigler, 2000). The execution of a model over time is understood as the simulation. While modeling targets the abstraction level (conceptualization), simulation challenges mainly focus on the implementation level. Interoperability allows exchanging of information between the simulations. Composability ensures the consistent representation of truth in all participating simulation systems of the federation. In other words, Interoperability is enabling interactions through common interfaces based on common conceptual models and shared context, whereas Composability is a shared understanding between components of their behavior. There are standards that focus on Interoperability, such as Basic Model Interface (BMI) (Hutton et al., 2020) and Open Modeling Interface (OpenMI) (Harpham et al., 2019).

The BMI is a library specification created by the Community Surface Dynamics Modeling System (CSDMS) to facilitate the conversion of a model or dataset into a reusable, plug-and-play component. Recall that, in this context, an interface is a named set of functions with prescribed arguments and return values. The BMI functions make a model self-describing and fully controllable by a modeling framework or application. By design, the BMI functions are straightforward to implement in any language, using only simple data types from standard language libraries. Also, BMI functions are noninvasive. This means that a model's BMI does not make calls to other components or tools and is not modified to use any framework-specific data structures. A BMI introduces no dependencies into a model, so the model can still be

used in a stand-alone manner. While a BMI can be written for any language, CSDMS currently supports five languages: C, C++, Fortran, Java, and Python. BMI allows the model code to be externally controlled and the variables to be exchanged at runtime ("BMI-2 Documentation", 2022).

Jiang et al. (Jiang et al., 2017) published environmental models as RESTful web services using the BMI and then integrated these services within the CSDMS framework. The Componentized WEAP RESTful framework will be explained in the next chapter. Comparing BMI with our research, the BMI functions have the same role as Componentized WEAP services. Also, the Componentized WEAP REST APIs are comparable with the web service-based version of the BMI (Jiang et al., 2017). There is one-to-one mapping between some of the BMI functions and the Componentized WEAP REST APIs (consequently, with the WEAP APIs). Some BMI functions do not have any equivalent API in the WEAP system due to WEAP's characteristics/attributes for water modeling and simulation. Different WEAP entities with their input and output variables and their data can be extracted and mapped to the BMI functions. The WEAP models are developed in 2D geographically, so the BMI functions regarding the Z axis are not meaningful in the WEAP system.

The OpenMI is an interface standard and consists of a core group of requirements and optional extensions. The core is fragile and defines the requirements for describing components and the data they can exchange, linking, and exchanging data; extensions deal with the more sophisticated data exchange requirements, such as the TimeSpace extension. The purpose of the core and extension concept is to allow for the future incremental development of the OpenMI. The original intention behind OpenMI was to provide a standard method that could be applied to independent numerical model components and allow them to exchange data while running.

The key feature of the standard is to enable the creation of links between components, where a link matches a variable in one component with its equivalent in another. These variables are referred to as either input or output exchange items. Related to the links are the GetValues and SetValues calls. These calls enable components to obtain/get the values of a variable from one component or change/set them in another. Bi-directional links are also possible. In the OpenMI context, the Adaptor element handles unit transformations and differences in model temporal and spatial transformations (e.g., vector/raster/non-spatial).

Many existing works tried to dynamically exchange data among web services using a legacy component-based framework. For example, Goodall et al. (Goodall et al., 2011), Castronova et al. (Castronova et al., 2013), and Gao et al. (F. Gao et al., 2019) developed OpenMI-compliant components to act as a client that is responsible for handling data transfers to and from the model service and then integrated the component using OpenMI framework.

3.6 Simulation Verification & Validation

Verification and Validation (V&V) are two distinct but complementary processes in modeling and simulation (Helton, 1993). Verification ensures that a simulation model is implemented correctly and free of errors (Yaung et al., 2014). Validation ensures that the model is an accurate representation of the real-world system being simulated (Oberkampf & Trucano, 2002; Roache, 1998). Both verification and validation are processes that accumulate evidence of a model's correctness or accuracy for some specific scenarios; thus, V&V cannot prove that a model is correct and accurate for all possible scenarios, but, rather, it can provide evidence that the model is sufficiently accurate for its intended use (Thacker et al., 2004). It is important to note that verification and validation are ongoing processes that should be conducted

throughout the development of the simulation model (Sargent, 2010). Quantifying the confidence and predictive accuracy of model calculations provides the decision-maker with the information necessary for making high-consequence decisions. The expected outcome of the model V&V process is the quantified level of agreement between experimental data and model prediction, as well as the predictive accuracy of the model.

Model V&V fundamentally differs from software V&V. Code developers developing computer programs perform software V&V to ensure code correctness, reliability, and robustness. In model V&V, the end product is a predictive model based on the fundamental physics of the problem being solved. In all applications of practical interest, the calculations involved in obtaining solutions with the model require a computer code. Therefore, engineers seeking to develop credible predictive models critically need model V&V guidelines and procedures (Thacker et al., 2004).

There are different methods to verify and validate a simulation model, such as:

- *Code review*: This method involves a thorough examination of the simulation code to ensure that it is correct and that it accurately represents the system being simulated (Kenney, 2017).

- *Unit testing*: This method involves the testing of individual components of the simulation code to ensure that each component functions correctly (Kenney, 2017).

- *Software testing*: This method involves the testing of the simulation code as a whole to ensure that it functions correctly and produces the expected output (Kenney, 2017).

- *Sensitivity analysis*: This method involves varying the inputs to the simulation model and analyzing the resulting outputs to determine the sensitivity of the

model to changes in input parameters (Helton & Davis, 2003; Wainwright & Sivilotti, 2018).

- *Statistical analysis*: In this method, statistical tests are used to verify the model. The output of the model is compared with the statistical properties of the system, such as mean, variance, and correlation. If the output of the model matches the statistical properties of the system, then the model is said to be verified (Law & Kelton, 2000).

- *Analytical V&V*: In this method, the model is verified and validated using mathematical analysis or analytical techniques. This method is useful when the model has a closed-form solution or when the behavior of the system can be predicted mathematically (Bhatnagar & Narang, 2014).

- *Expert review*: This method involves the review of the simulation model by experts in the field to ensure that it accurately represents the system being simulated (Banks, 1998; Khosrowjerdi, 2020).

- *Cross-Validation*: In this method, a dataset is split into two or more subsets, and the model is trained on one subset and tested on the other subset. This helps to estimate the performance of the model on unseen data (Kohavi, 1995).

- *Experimentation*: Validation can also be achieved through experimentation, in which the simulation model is compared with data obtained from experiments conducted on the real-world system being simulated. If the experts agree that the model is correct, accurate, and relevant, then the model is said to be verified (Wainwright & Sivilotti, 2018).

- *Visual Inspection*: In this method, the data is visually inspected for any anomalies or inconsistencies. This can help to identify data errors, such as incorrect labels or mislabeled data (Tukey, 1977).

Chapter 4

WEAP AND LEAP COMPONENTIZATION DESIGN & IMPLEMENTATION

The WEAP and LEAP systems are used as the primary tools for modeling and simulating water and energy management under different socio-economic and climate scenarios for regions including South Africa, China, Greece, Benin, and Pakistan (Amin et al., 2018; J. Gao et al., 2017; Höllermann et al., 2010; Lévite et al., 2003; Psomas et al., 2016). This suggests that the WEAP and LEAP system's componentization can help combine these and other simulations with separate simulations for agriculture, climate, and land systems. Given the related works highlighted before, the Componentized WEAP and LEAP are tools built using the RESTful SOA architecture, and the data cube structure (see Figure 2.8) allows integrating it with other tools for simulating the FEW-Nexus.

In the rest of the dissertation, a simple WEN example is used to exemplify the concepts and ideas. The WEN example is presented in the following frame to make it distinguishable from the conceptual sections.

WEN Example

As a simple example, Figure 4.1 illustrates a schematic water-energy model of a prototypical region. Each part in this water-energy model has a level of detail given its exact counterparts. Given their use for understanding and supporting decision policies, the water and energy models satisfy some primary, high-level constraints individually and relative to one another. In Figure 3, the "Farm" needs $0.2hm^3$ ($1hm^3 = 1,000,000m^3$) of water and $5GWh$ of electricity per year. The required water for the "Farm" is supplied from an aqueduct (i.e., "Canal_2") through a pumping and

treating station with an energy intensity of $7kWh/m^3$ (i.e., "Pump_2"). The required energy for the whole model is generated by a coal power plant that diverts (for its operation) water from the river (i.e., "Canal_1") through a pumping and treating station with an energy intensity of $4kWh/m^3$ (i.e., "Pump_1"). The objective is to model the interactions between the water and energy systems to satisfy their constraints. The water system has two suppliers and two demands, and the energy system has one supplier and three demands (the "Farm" is a demand in both water and energy systems).

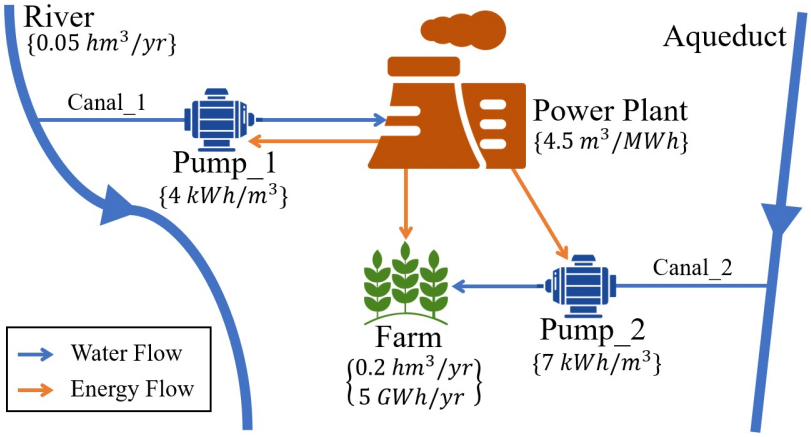


Figure 4.1: A Schematic View of a Region with Common Types of Water and Energy Systems.

Suppose having seasonal time granularity for the water mode and yearly time granularity for energy model, the Formulas 4.1, 4.2, and 4.4 show the data transformations between the systems. The year property can be between the start-year and end-year of the simulation. The required energy for "Pump_1" equals the sum of the Flow in "Canal_1" in all seasons of a year, multiplies by 0.004 (see Figure 4.1). The same formula applies for the required energy in "Pump_2", but with different constant factors (i.e., 0.007). Finally, formula (4) shows that the required water for the "Power Plant" entity is calculated by dividing the generated electricity by four (number of seasons in a year), and then multiplies by 4.5 (see Figure 4.1).

$$Required\_Energy^{year}_{Pump\_1} = (\sum_{1 \leq ts \leq 4} Flow^{ts,year}_{Canal\_1}) \times 0.004 \tag{4.1}$$

$$Required\_Energy^{year}_{Pump\_2} = (\sum_{1 \leq ts \leq 4} Flow^{ts,year}_{Canal\_2}) \times 0.007 \tag{4.2}$$

$$\begin{aligned} Generated\_Electricity^{year}_{PowerPlant} = Required\_Energy^{year}_{Pump\_1} + \\ Required\_Energy^{year}_{Pump\_2} + Required\_Energy^{year}_{Farm} \end{aligned} \tag{4.3}$$

$$Required\_Water^{year}_{PowerPlant} = (\frac{Generated\_Electricity^{year}_{PowerPlant}}{4}) \times 4.5 \tag{4.4}$$

Figure 4.2 illustrates the developed models in the WEAP and LEAP tools for the presented water and energy systems in Figure 4.1 (suppose ignoring the different time granularity in the systems and using the WEAP-LEAP internal linkage). The water model has two *River*, two *Transmission*, and two *Demand Site* entities. The energy model has one *Resource*, one *Transformation*, and three *Demand* entities. Based on the Formulas 4.1, 4.2, and 4.4, the solid red (reading *Electricity* values) and dotted blue (reading *Flow* values) arrows between the water and energy models present the direct inter-connection among the systems (without using a separate interaction model).

The "PowerPlant" demand site entity in the WEAP model needs to know the amount of generated *Electricity* by the "PowerPlant" transformation entity in the LEAP model. The "Pump1" and "Pump2" demand entities in the LEAP model need to know the amount of *Flow* in the "Canal1" and "Canal2" transmission link entities in the WEAP model. The required amount of water for the "Farm" demand site entity in the WEAP model (which is $5000 kWh/year$), and the required amount of energy for the "Farm" demand entity in the LEAP model (which is $200,000 m^3/year$)

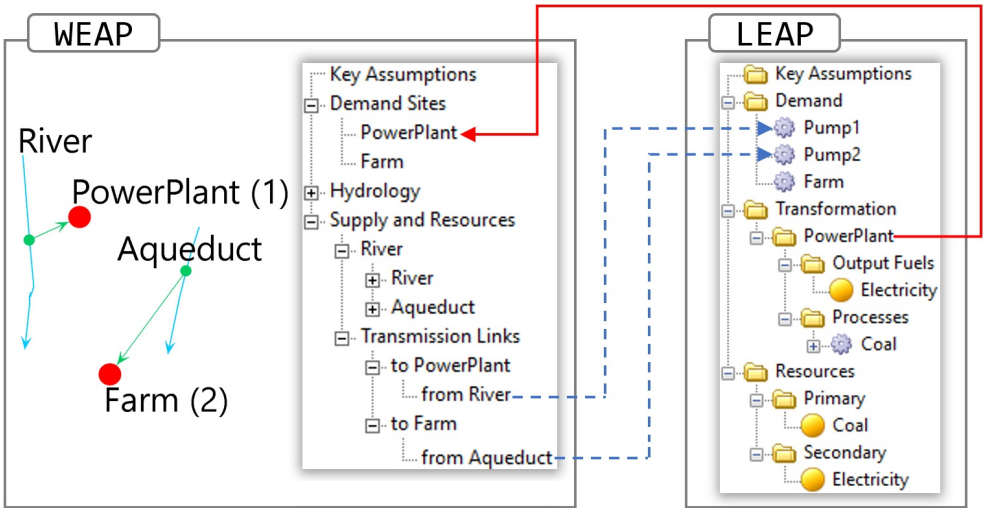


Figure 4.2: A Defined Model for the Exemplar Wen Model in the WEAP and LEAP Systems and Their Direct Inter-connections.

are set as constants in the corresponding models. Using the data-sharing approach, the formulas and constraints presented in Figure 4.1 must be calculated and satisfied inside the WEAP and LEAP models (usually in the target system). In other words, the energy model must read the *Flow* values from the water model to apply the conversions and check the constraints (the same scenario for reading *Electricity* values from the energy model by the water model). In addition to the constraints in using the WEAP-LEAP internal linkage mechanism, changing and maintaining the WEN model would be cumbersome due to the direct and strong dependency between the connected models.

4.1 Web-Service Framework for the WEAP System

According to the constraint for using the JavaScript language (to invoke Automating the use of WEAP APIs (SEI, 2022d)) and the difficulties of using XML-based protocols (extensive code development to create XML structure) (Tihomirovs & Gra-

bis, 2016), the RESTful framework is used to implement the web-service framework for the WEAP system.

4.1.1 Models of the WEAP Entities

To componentize the WEAP system, the entities that are included in the WEAP system with their data are mapped to components using the Ecore Modeling Framework (EMF) (Budinsky et al., 2004; Steinberg et al., 2008). The Ecore meta-model is used to model the WEAP entities at an abstract view without specifying their functions. At this abstraction level, the data structure of different WEAP sections, entities, variables, and the relationship among these parts are modeled. The specification in Figure 4.3 is defined using the EClass, EAttribute, EDataType, and EReference elements of the Ecore meta-model diagrams. It is important to note that the WEAP's APIs expose the scope and functionality of the componentized entities defined for the WEAP framework (SEI, 2022a). For example, it is not possible to add new variables for any entity via WEAP's API; thus, adding these variables must be achieved by the user within the WEAP system (see Figure 2.5b).

In Figure 4.3, the WEAP class has various projects, each associated with a geographic area. Each `Project` has its configuration (*name*, *startYear*, *endYear*, *timeStepPerYear*, and so on), which are mapped to the properties in Figure 2.4. The Node and Link are two abstract classes that are detailed in the following sections. The `WEAP`, `Project`, `Version`, and `Scenario` are concrete classes. These correspond to the entities in the WEAP system which instantiate a specific domain model in the WEAP RESTful framework. The remaining abstract and concrete classes are helpful for the design to be simple yet flexible.

A simulation model in the WEAP system has a structure defined by the modeler, but the behaviors for the specialized node and link entities are predefined. The date

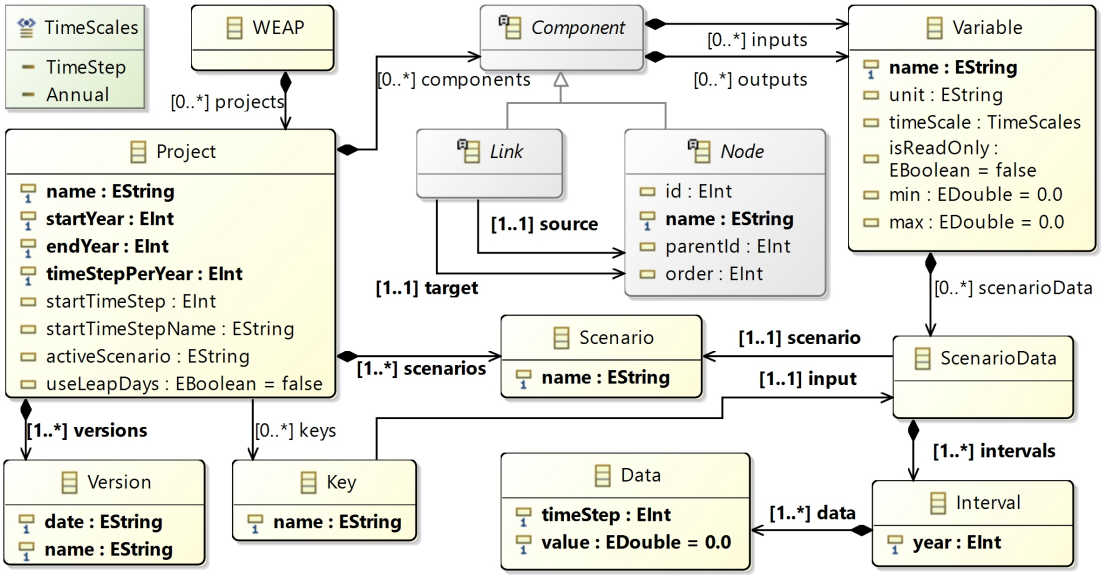


Figure 4.3: Ecore Specification to Model WEAP's Entities, Variables, and Data.

specified in a scenario is needed to simulate some aspects of a water system. Every project has at least one scenario, the *Current Accounts*, which provides a snapshot of actual water demand, pollution loads, resources, and supplies for the system (Sieber et al., 2005). It is used to define the initial data for the inputs at the start of a simulation. The hierarchical structure of scenario data in the WEAP system does not have any functional role. Each data for a scenario is independent of any other scenario data (see our perspective about the data in Figure 2.8). Also, every project has at least one version (refer to the creation time of the project). The name property is the key attribute in the `Project`, `Scenario`, `Variable`, `Key`, and `Node` classes (see Figure 4.3). The key for the `Version` class is the concatenation of its properties (*date* and *name*). The Componentized WEAP RESTful framework has a complete set of model components that cover all the entities and variables in the Schematic, Data, and Result views of the WEAP system (see Figure 2.5). The derived model components do not add any operations to those provided for the WEAP entities. The

model components are categorized into the Node and Link types according to their properties (see Figure 4.3). Each Link has one source node and one target node.

All model components have some input and output variables (see Figure 4.3). The WEAP system has some predefined entity variables and equations. New variables and equations may be added by users as needed. For each variable, one or more intervals are defined per scenario, and each interval can have many data values (a value represents a specific time-step of a year). The Variable class has a unique name property as the key with the *unit*, *timeScale*, *isReadOnly*, *min*, and *max* properties. A variable can have just one value per year if the *timeScale* property sets to "Annual", or it can have multiple values (the number of values should be equal to the *timeStepPerYear* property in `Project` class) if *timeScale* sets to "TimeStep". The value of an input variable cannot change if the *isReadOnly* property is set to "True". The properties *min* and *max* place constraints on the acceptable values for a variable. From a higher abstraction view shown in Figure 4.3, the `Node` and `Link` classes with their input and output variables define the structure of a model.

The Componentized WEAP RESTful framework has the same schema (a generic view) for all the WEAP entity types (e.g., *Demand Site*, *Catchment*, and *Transmission Link*) and their variables. For example, a catchment can have a different set of input variables based on its selected simulation method (such as *Rainfall Runoff*, *Irrigation Demand Only*, and *MABIA*), and the Componentized WEAP framework presents a set of input variables (see Figure 4.3) for this entity at a high abstraction level. Thus, two catchments in a project can have different sets of input variables. The `Variable`, `ScenarioData`, `Scenario`, `Interval`, and `Data` classes with their relations define the overall input data and output result for a model in the Componentized WEAP RESTful framework that mirrors those defined in the WEAP system.

As shown in Figure 4.4, different nodes (correspond to the entities in Figure 2.5a) are inherited from `Node`, `Flow`, or `ReachPoint` abstract classes. The `River` and `Diversion` nodes have some sub-nodes, which are an ordered collection of reach point nodes. Consequently, a variable of a flow entity can have different values in its reach points. As shown in Figure 4.5a, the WEAP system has three link entities (*Transmission Link*, *Return Flow*, and *Runoff*). Each link starts from a node and ends at another node with some constraint for the source and target nodes based on the link type (Sieber et al., 2005). The allowable source and target nodes for the `Transmission Link`, `Return Flow`, and `Runoff` classes are shown in Figures 4.5b, 4.5c, and 4.5d, respectively. Each of the entities has its input and output variables and data (see Figure 4.3).

4.1.2 Mapping Componentized WEAP Models to a RESTful framework

The model components of the WEAP system are the actual resources in the Componentized WEAP RESTful framework, so a well-defined structure for the URL needs to be present to operate on the resources (ACIMS, 2022b). The data needed for the RESTful framework is in JSON format. The RESTful API categories are *Project*, *Version*, *Key*, *Node*, *Link*, and *Flow*. In the pattern of the URLs, constants are written in `PascalCase` style; parameters start with colons and are written in `camelCase` style; query parameters (to apply to some filters on returned data) written after the question mark by Key=Value (`camelCase` style for the Key part). The retrieve, insert, update, and delete operations for each URL are supported with the HTTP GET, POST, PUT and DELETE methods, respectively.

The URL patterns for six API types are shown in Table 4.1. The pattern inside each open and close pair bracket is optional. The appropriate types are presented in Table 4.2. There is a mapping between the URL patterns in Table 4.1 and the

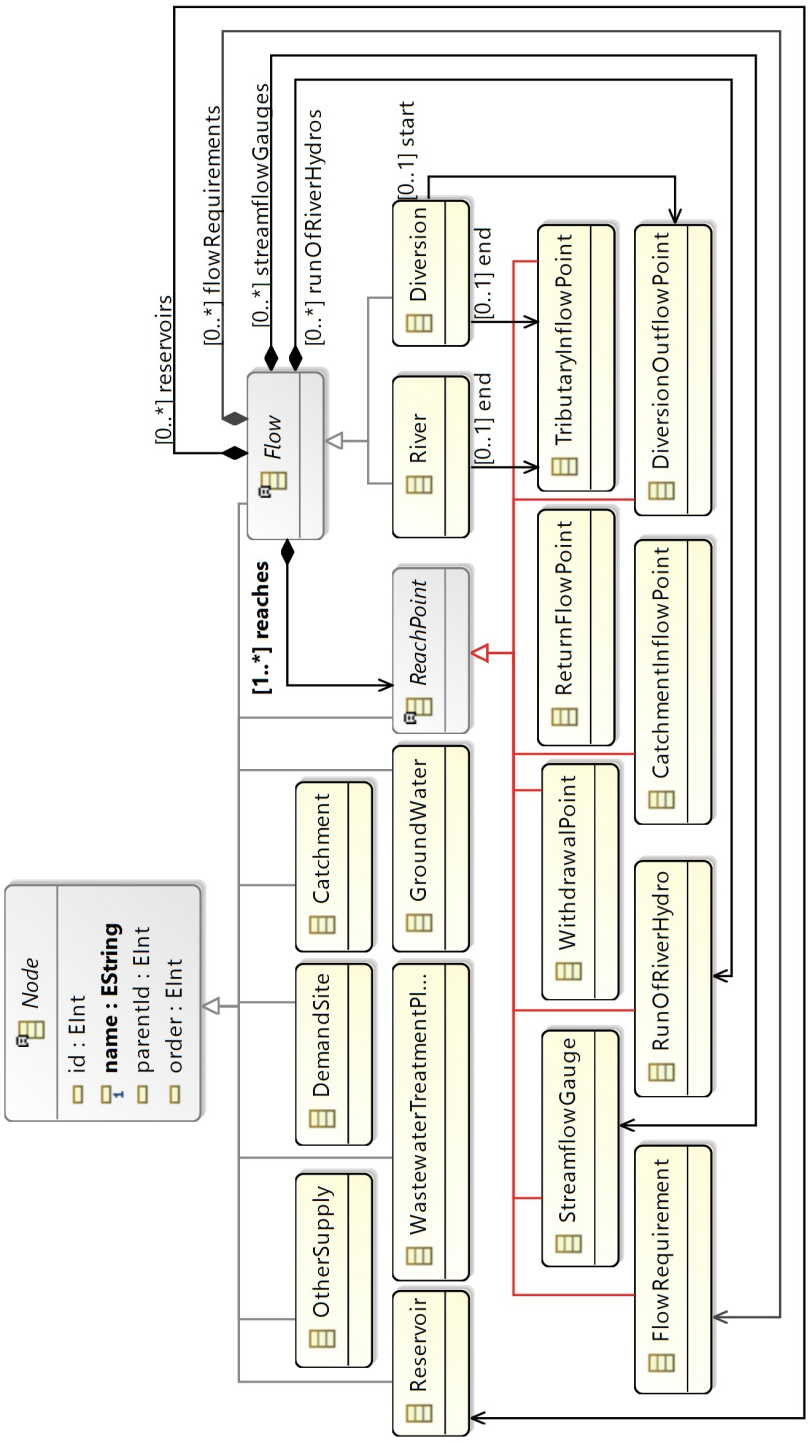
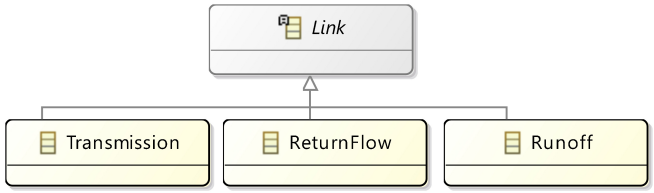
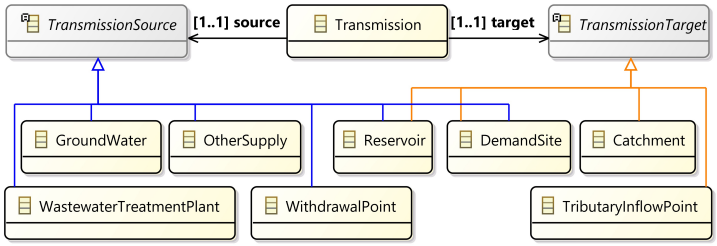


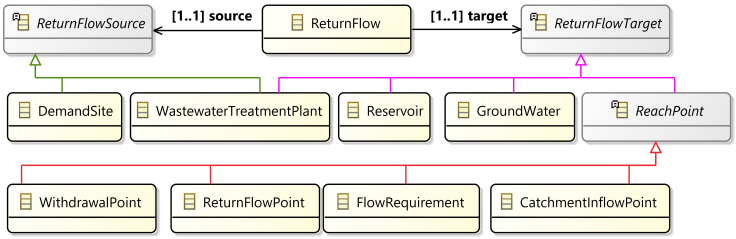
Figure 4.4: Ecore Specification for the Node Entities in the WEAP RESTful Framework.



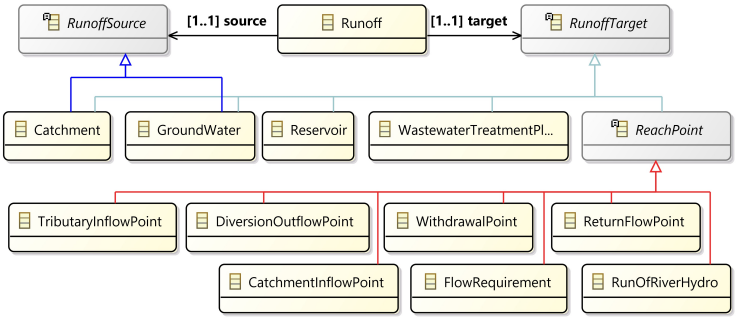
(a)



(b)



(c)



(d)

Figure 4.5: Specified Ecore Models for the WEAP Link Entities. (a) Three Link Types in a WEAP Model. (b) Source and Target Nodes for a *Transmission Link* Entity. (c) Source and Target Nodes for a *ReturnFlow* Entity. (d) Source and Target Nodes for a *Runoff* Entity.

Ecore specifications in Figure 4.3, Figure 4.4, and Figure 4.5. All URLs start with the constant "/Water", which refers to the `WEAP` class shown in Figure 4.3. For example, calling "/Water" returns the name of all projects (an array of string) corresponding to the composite relation from the `WEAP` class to the Project class in Figure 4.3. When a project is selected using the `:projectName` parameter, the project configuration information can be read or changed depending on the URL's method. Finally, a model (e.g., `demo`) can be executed using the URL "/Water/demo/Run". The `Version` saves the project in different timestamps. The list of versions can be retrieved using the URL "/Water/demo/Version". The project reverts to a specific version (e.g., 20200820-test) using the URL "/Water/demo/20200820-test/Revert".

In the patterns for the Node, Link, or Flow categories in Table 4.1, the constant types must be replaced by one of the values in its corresponding type in Table 4.2. The name of a Node or Flow, and the names of the source and target nodes of a Link are used to select a component. For example, the URL "/Water/demo/ DemandSite/phoenix" returns the phoenix demand site's data of the `demo` project. The `VariableType` in the URL patterns must be replaced by the "*Inputs*" or "*Outputs*" (refer to the Data or Result variable in the WEAP system). The data of a variable can be retrieved by mentioning the name of the variable and the intended scenario. The expression of a variable can be retrieved by adding the Expression constant in the URL. Query parameters can be used to filter the returned data (the years and time-steps). In the Flow URLs, the `subNodeType` must be replaced with the corresponding value in Table 4.2 (the reference properties in `Flow` abstract class in Figure 4.4) to access a specific collection of sub-nodes, and then use `:subNodeName` to select one.

The "*Weaping River Basin*" project is one of the default projects in the WEAP system, with 12 time-steps per year from 2010 to 2020 (Yates et al., 2005). Figure 2.5a

Table 4.1: URL Signatures for Different Types of Componentized WEAP APIs.

Category	URL Signatures
Project	/Water[/:projectName[/Run]]
Version	/Water/:projectName/Versions[/:versionName/Revert]
Key	/Water/:projectName/Keys[/:KeyName/: scenarioName[/Expression]]
Node	/Water/:projectName/NodeType[/:nodeName[/VariableType[/: variableName/:scenarioName[/Expression][?startYear=N& endYear=N&startTimeStep=N&endTimeStep=N]]]]
Link	/Water/:projectName/LinkType[/:sourceName/: targetName[/VariableType[/:variableName/: scenarioName[/Expressi on][?startYear=N&endYear=N& startTimeStep=N&endTimeStep=N]]]]
Flow	/Water/:projectName/FlowType[/:flowName[/subNodeType[/: subNodeName]][/VariableType[/:variableName/: scenarioName[/Expression][?&startYear=N&endYear=N& startTimeSt ep=N&endTimeStep=N]]]]

shows the Schematic View of this project in the WEAP system. Figure 4.6a shows the result of calling an API to get the rivers in the *Weaping River Basic* model. The returned data is an array that contains three River objects. Figure 4.6b shows the result of calling an API to get the data for the *Annual Activity Level* variable of the *West City* demand site for the *Reference* scenario between 2010 and 2012. The returned data is an array of Interval objects. The variable in this example has a yearly time-scale, so the result has one instance of Data class (one pair of *timeStep* and *value*). Using a variable with the *TimeStep* time scale will return 12 (due to the

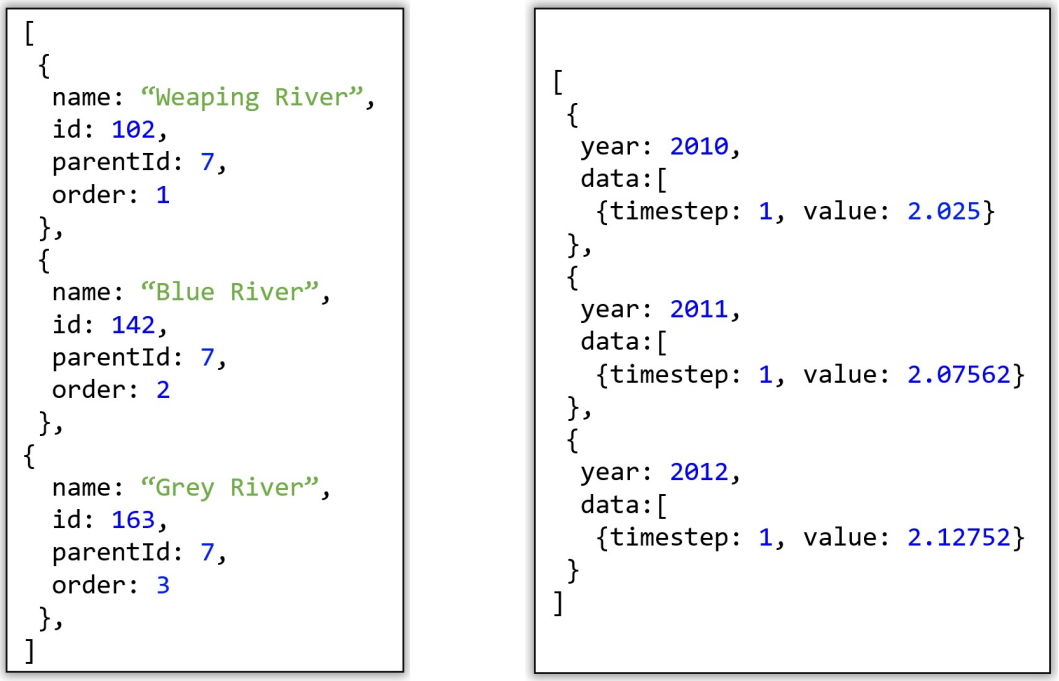
Table 4.2: Constants in the URL Signatures of Table 4.1.

Type	Value
NodeType	`Catchments`, `DemandSites`, `Groundwaters`, `Reservoirs`, `OtherSupplies`, `WastewaterTreatments`
LinkType	`Transmissions`, `Runoffs`, `ReturnFlows`
FlowType	`Rivers`, `Diversions`
VariableType	`Inputs`, `Outputs`
subNodeType	`Reaches`, `Reservoirs`, `RunOfRiverHydros`, `StreamflowGauges`, `FlowRequirements`

timeStepsPerYear value of the project in Figure 2.4) instances of Data class for each year. Tree structures for the Node and Interval shown in Figure 4.6 are generated by the Componentized WEAP RESTful framework.

4.1.3 Design and Implementation

The layered architecture of the Componentized WEAP RESTful framework is shown in Figure 4.7. The dotted area indicates the server-side layers of this framework. The "WEAP" system and the "File System" are placed at the framework's bottom layer. The externalized WEAP model components and their configurations are stored in a CSV file. The "Data Access Objects" layer is responsible for ensuring the consistency of the componentized models and their configurations at all times with its WEAP system counterpart. Only this layer has direct access to the "File System" and to the "WEAP" system via its APIs (SEI, 2022d). It can communicate with any model that exists in the bottom layer (i.e., for creating and/or executing WEAP models). The frameworks are supported by a local catching mechanism to increase the performance of structure-related requests. It means that, given receiving



(a) (b)

Figure 4.6: Result of Calling Componentized WEAP APIs. (a) Result of Calling the URL= “/Water/Weaping River Basin/Rivers”. (b) Result of Calling the URL= “/Water/Weaping River Basin/DemandSites/West City/Inputs/Annual Activity Level/Reference?startYear=2010&endYear=2012”.

a request related to the structure of a model, the catching system is first checked. Suppose the data is not in the catching system. In that case, it will be fetched from the WEAP system and inserted into the catching system for the following requests.

All communications between the “Data Access Objects” and the “Web APIs” layers are managed by the “Data Transfer Object” and “Service” parts (see Figure 4.3, Figure 4.4, and Figure 4.5). The “Service” layer is responsible for communicating and processing information about the WEAP entities (contained in the bottom layer) via the componentized “Data Access Object” layer to the “Web APIs” layer. Every Componentized WEAP model (which is identical to the WEAP system entities) can be manipulated by an “API Caller” in an independent fashion. The “Web APIs” layer

contains the web-server and controller parts (not shown in Figure 4.7) for handling various client API requests.

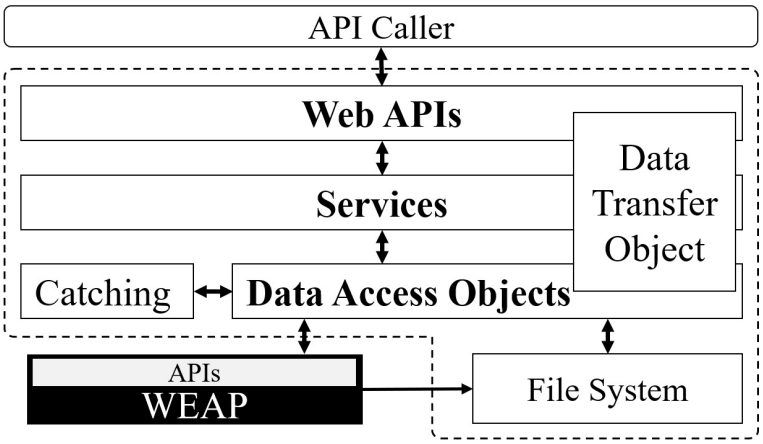


Figure 4.7: Componentized WEAP RESTful Framework Layer Architecture.

Figure 4.8 illustrates the major packages of the Componentized WEAP RESTful framework. The `src` package contains the packages that correspond to the layers in Figure 4.7. The `utilities` package contains the `constants`, `enums`, `helpers`, and `factories` packages. The `controllers`, `services`, and `dataAccess` packages contain concrete classes and a sub-package for their interfaces.

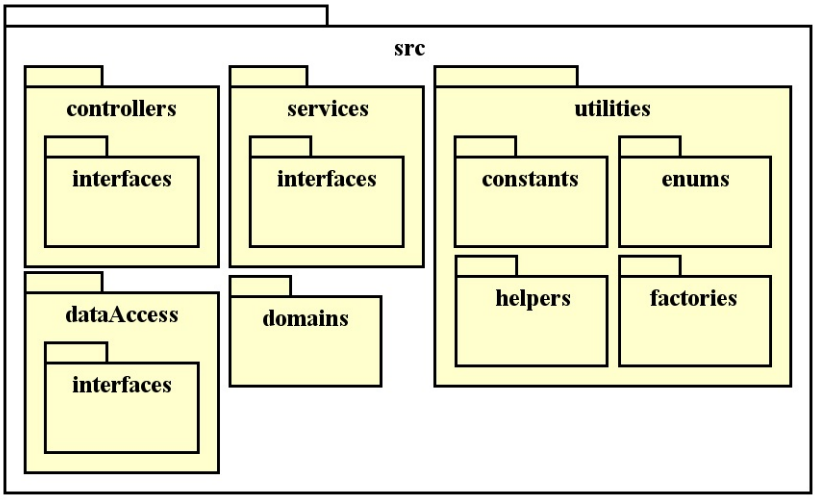


Figure 4.8: Componentized WEAP RESTful Framework Package Diagram.

Figure 4.9 shows the interfaces defined in the `controllers/interfaces` package in Figure 4.8. The `INodeController` is the super class for all node-type components (see Figure 4.4). It has 11 methods to get and/or set the properties of the node, input and output ports/variables, and their relevant data and/or defined expression. The `ILinkController`, the super class for all link-type components, has the same methods as defined in the `INodeController`, except a specific component can be addressed using source and target names (instead of component name). For example, "`getInputs(projName: string, source: string, target: string): Variable[]`" defines to return all inputs of a link started from a source node and ending at a target node. The `IFlowController`, the superclass for all components of type *Flow*, is inherited from `INodeController`. It also has the same set of methods (defined in the `INodeController`) for sub-nodes components (`ReachPoints`, `Reservoirs`, `RunofRiverHydros`, `StreamflowGuages`, and `FlowRequirements`; see Figure 4.4). The Water (referring to the `WEAP` class in Figure 4.3), `Project`, `Scenario`, `Version`, and `Key` controllers have their specific interfaces.

The `services/interfaces` package has the same set of interfaces (by changing the term "Control" to "Service" in the name of the interfaces). Indeed, controllers are responsible for getting the incoming requests on the web-server, validating their parameters, and calling the appropriate methods from the service layer. A set of concrete classes (as the controllers) are defined in the `services` package. Each controller implements its corresponding interface and has a relation to the relevant service.

Figure 4.10 presents the classes defined in the domains package of the Componentized WEAP RESTful framework to transfer data between different layers, and define the structure of the returned data to the API's caller. The `Component` class in Figure 4.10 is an abstract class and the rest are concrete classes. This is a realization of the Ecore specification defined in Figure 4.3, Figure 4.4, and Figure 4.5. There

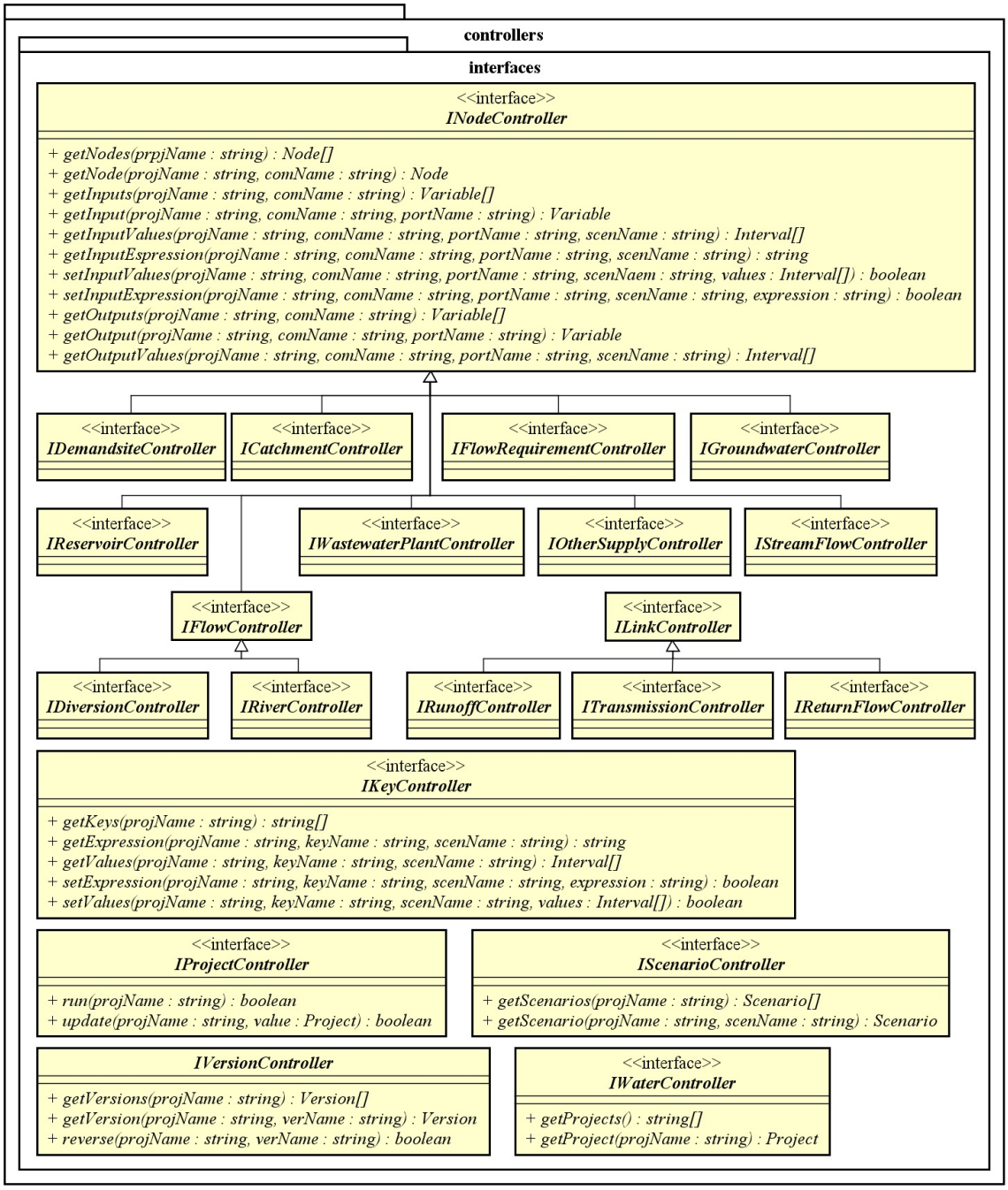


Figure 4.9: A Class Diagram of the Web APIs Interfaces of the Componentized WEAP RESTful Framework.

are two composite relations in Figure 4.10 from the `Water` class to the `Project` class and the `Interval` class to the `Data` class. Also, the source and target nodes for a link are defined using two association relations from the `Link` class to the `Node` class.

For brevity, the *set(...)* and *get()* operations are excluded from the classes in the diagram.

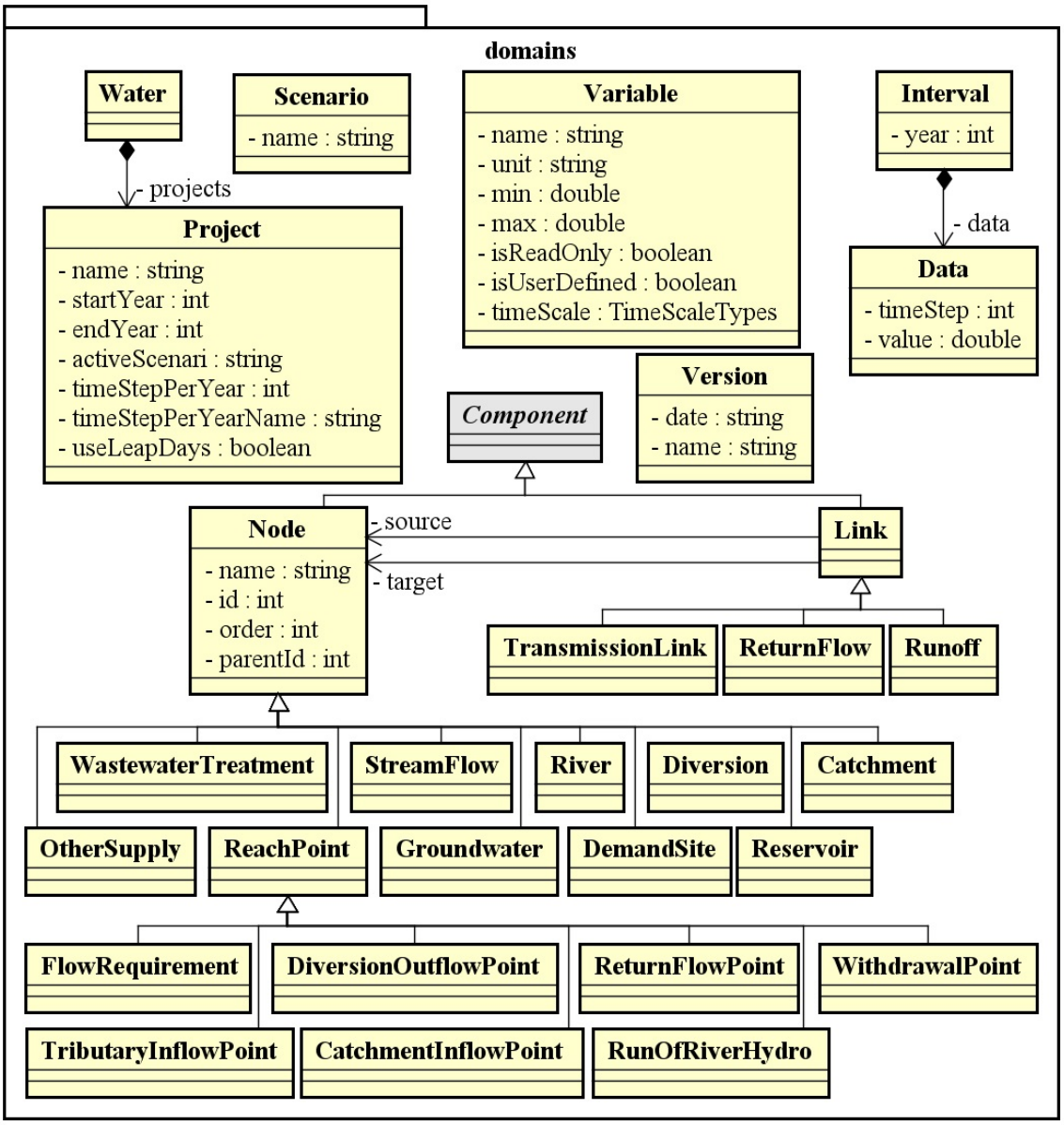


Figure 4.10: A Class Diagram of the Data Transfer Object Models of the Componentized WEAP RESTful Framework.

Figure 4.9 shows the interfaces defined in the `dataAccess/interfaces` package. The data access is designed based on a Fluent Interface design pattern. A Fluent Interface is an object-oriented API that relies extensively on method chaining and pro-

vides more readable code. Thus, the "`Inputs()`" method in the `IAbstractNodeDao` in Figure 4.9 returns a list of variables, but the "`Input(name: string)`"returns `IVariableDao` object (by setting the port name). The `NodeX`, `LinkX`, and `FlowX` in the method names of the `ProjectDao` must be replaced with corresponding node, link, and flow entity types. As an example, Listing 4.1 contains two methods that present how to set an input port expression, and how to get an output port data in the `DemandSiteService` class. Calling the service methods always starts from `WEAPDao` object, then goes to the `ProjectDao` object, and so on (in getting the output example, going to the `DemandSite`, then `Output`, and finally calling the `get()` method).

Listing 4.1: `setInputExpression` and `getOnput` Methods in the `DemandSiteService` of the Componentized WEAP RESTful Framework.

```
setInputExpression(projName: string, compName: string, portName:
   string, scenName: string, value: string): boolean {
     boolean res = new WEAPDao()
          .Project(projName)
          .DemandSite(compName)
          .Input(portName)
          .Value(scenName)
          .setExpression(value);
return res;
}

getOutput(projName: string, compName: string, portName: string):
   Variable {
  Variable res = new WEAPDao()
       .Project(projName)
       .DemandSite(compName)
       .Output(portName)
       .get();
Return res;
```

```
18 }
```

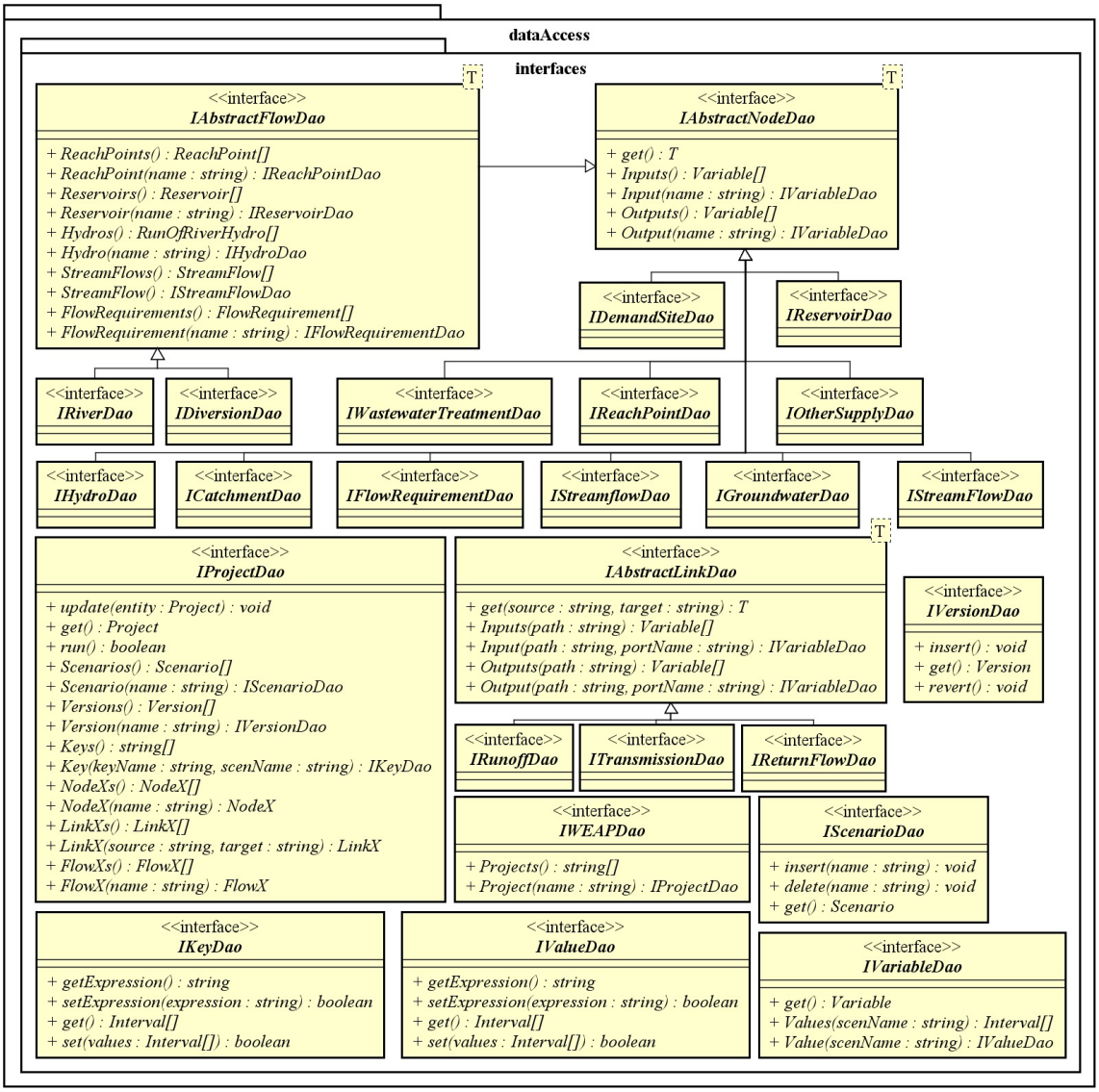


Figure 4.11: A Class Diagram of Interfaces in the "Data Access Objects" Layer in the Componentized WEAP RESTful Framework.

A partial class diagram for the concrete classes of the "Data Access Objects" layer is shown in Figure 4.12. This is a realization of the Ecore specification defined in Figure 4.3. The `AbstractNodeDao` is a generic abstract class, and the rest are concrete classes. Only one method is presented for each class in the diagram to show the required parameters of the class constructor. The `WEAPDao` class is the entry point

of the "Data Access Objects" layer to apply changes or retrieve information from the WEAP system. The `WEAPDao` just has the *weap* attribute, which instantiates a WEAP ActiveXObject (using `winax` library). The `WEAPDao` class implements the `IWEAPDao` interface, retrieving a list of all projects and retrieving a specific project (using the project name) by returning a `ProjectDao` object. The instantiated `weap` object in the `WEAPDao` class always must be passed to other classes as a constructor's parameter in addition to other parameters (e.g., the name of the project in the `ProjectDao` class or component type in the `VariableDao` class). The `ProjectDao` class implements the `IProjectDao` interface (see Figure 4.11). It has access to all nodes, links, flows, scenarios, versions, and so on. (only the `Demand Site`, `Reservoir`, `Scenario`, and `Version` are presented in the diagram). The `AbstractNodeDao` implements all functions defined in the `IAbstractNodeDao` interface in a generic structure, and generalized classes (e.g., `DemandSite`, `Catchment`, `Reservoir`, and so on) define the component type in their constructor. The relation between the classes is shown using the association relation between them. For example, a project is accessible from the `WEAPDao` class; a scenario, version, demand site, reservoir, etc. are accessible from the `ProjectDao`; the input and output of a specific component are accessible from the `DemandSiteDao` and `ReservoirDao` classes in Figure 4.12; the value of an input or output is accessible from the `VariableDao` class.

A sequence diagram scenario for a client fetching the rivers of the *Weaping River Basin* project is shown in Figure 4.13. This specification is devised to show a normal (positive) sequence of messages among a select set of objects instantiated from the classes shown in Figure 4.13. At the end of this scenario, the three rivers in the *Weaping River Basin* project are identified. The incoming message 1 (RESTful API request) by the `cl` object is processed by the `ctrl` object. Subsequently, in steps 2-4, the `svc`, `wDao`, and `weap` objects are created. In step 5, the `ctrl` object parses an

incoming request to extract some parameter of interest (e.g., a project name). Then, message 6 is invoked on the `svc` object to find all existing rivers. The `svc` object invokes message 7 on the `wDao` object. A `pDao` object is created and then returned to the `svc` object. The `svc` object invokes message 9 on the `pDao` object for identifying the rivers in the project. The `pDao` object invokes message 10 on the `weap` object, which in turn finds the data for the rivers (SEI, 2022a). In the loop section, the `riv` objects are created in step 11 using the returned array in step 10 (see Figure 4.4). The properties of the `riv` object for each branch are updated in step 12. Finally, the list of the created rivers is returned to the `svc`, `ctrl`, and `cl` objects. This scenario depicts a complete cycle starting from a client application to the WEAP system and ending at the client (see Figure 4.7).

The componentization of the WEAP system supports a higher degree of control for manipulating and simulating the water entity models (Fard & Sarjoughian, 2021a). The approach can help simplify the design of simulation experiments and optimization studies that can be difficult using the scripting languages supported in the WEAP system. The RESTful framework with the WEAP/LEAP componentization can lend itself to better support the development of customized tools. The Componentized WEAP framework represents the defined model in the WEAP and LEAP systems in a well-structured format. Figure 4.14 illustrates the data retrieving in the Componentized WEAP RESTful framework. The process steps for a request are shown by the numbers 1 to 5 in the diagram. The process starts with receiving a request from the "API Gateway". Based on received parameters (i.e., *Entity Type*, *Entity Name*, and so on), a proper method from the "Componentization" part is called to fetch data from a specific WEAP entity. Each entity has a list of variables (e.g., $Variable_1$ to $Variable_i$ of $Entity_1$ in Figure 4.14). Also, the values of a variable are specified according to the *year*, *timestep*, and *scenario* parameters (specified by

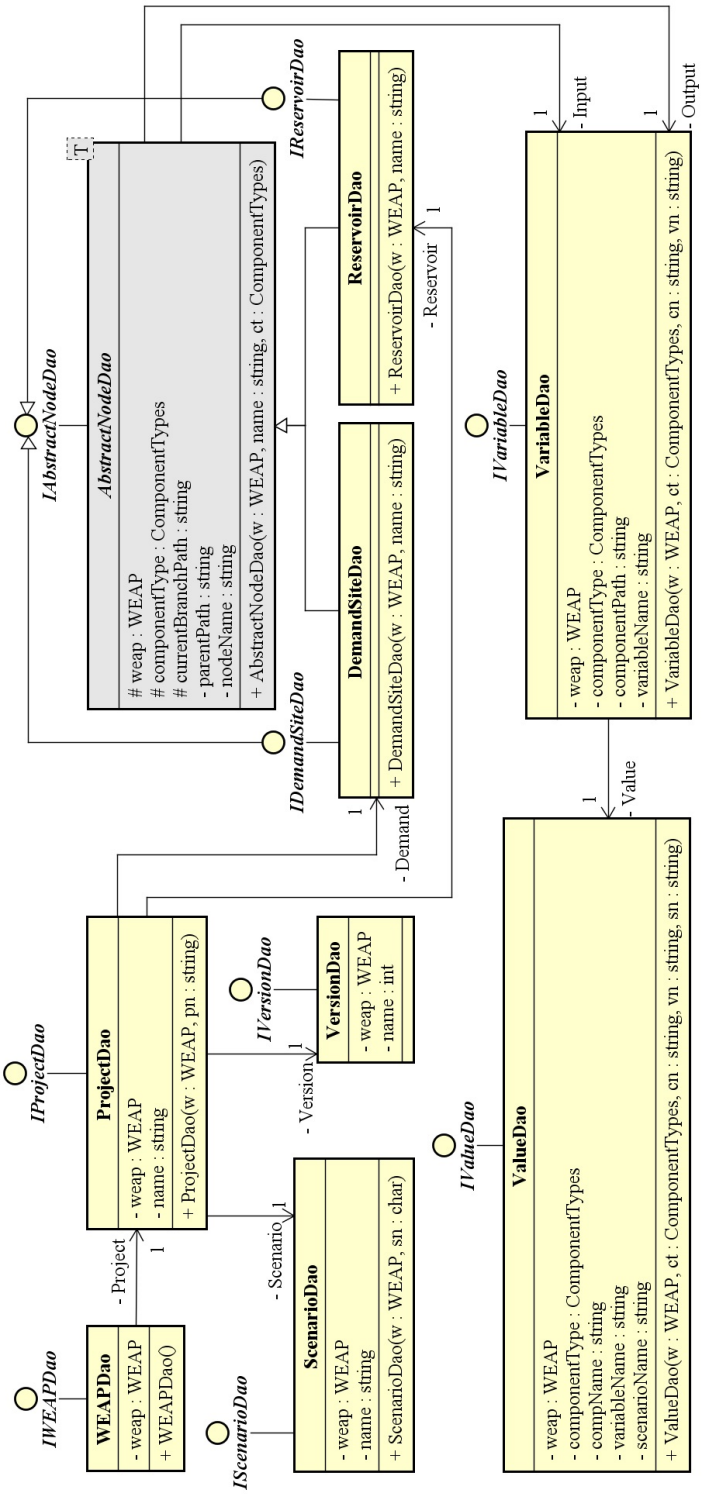


Figure 4.12: Class Diagram for the "Data Access Objects" Layer of the Componentized WEAP Framework.

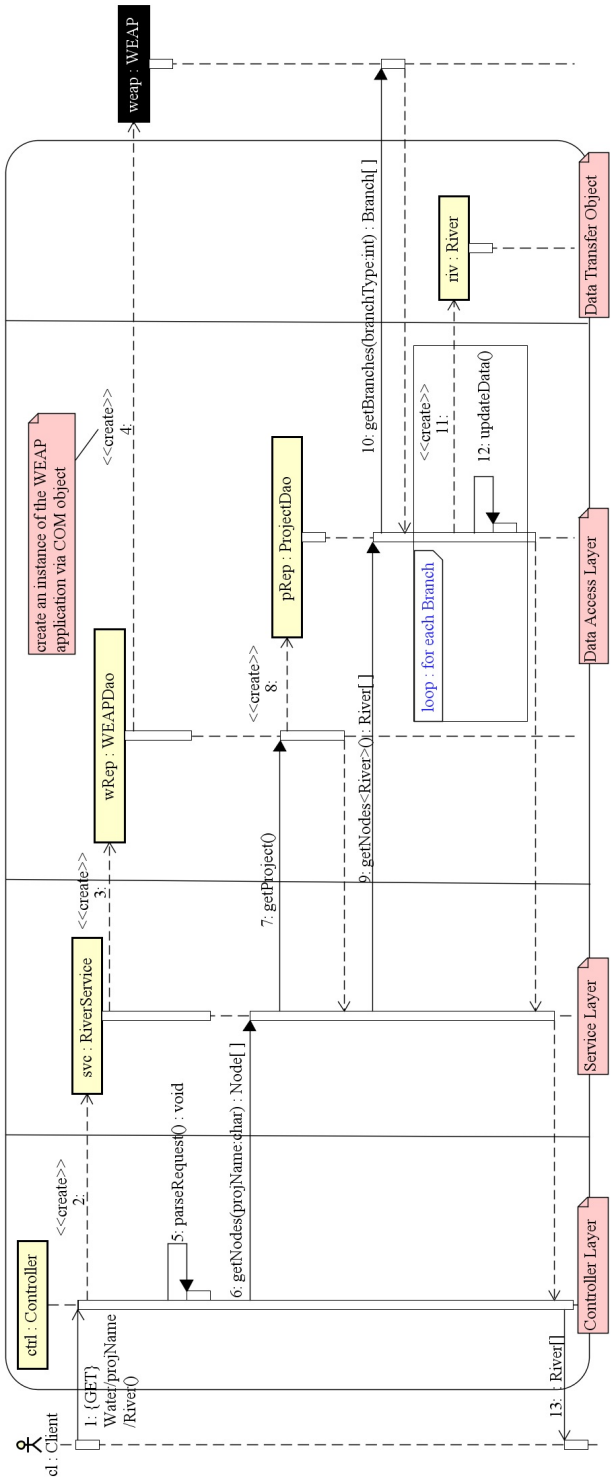


Figure 4.13: Sequence Diagram to Get All Rivers of a Project via the Componentized WEAP RESTful Framework.

(y_1,Δt_1,s_1) to (y_m,Δt_n, s_r) for each variable in Figure 4.14). The data specified by $d_{1,1}$ to $d_{m,n}$ are real data type values. The "Componentization" part retrieves the values of a variable one by one to create a list of values based on years and timesteps (step 4 in Figure 4.14). Finally, the values will be converted to JSON format and sent to the "REST API Caller".

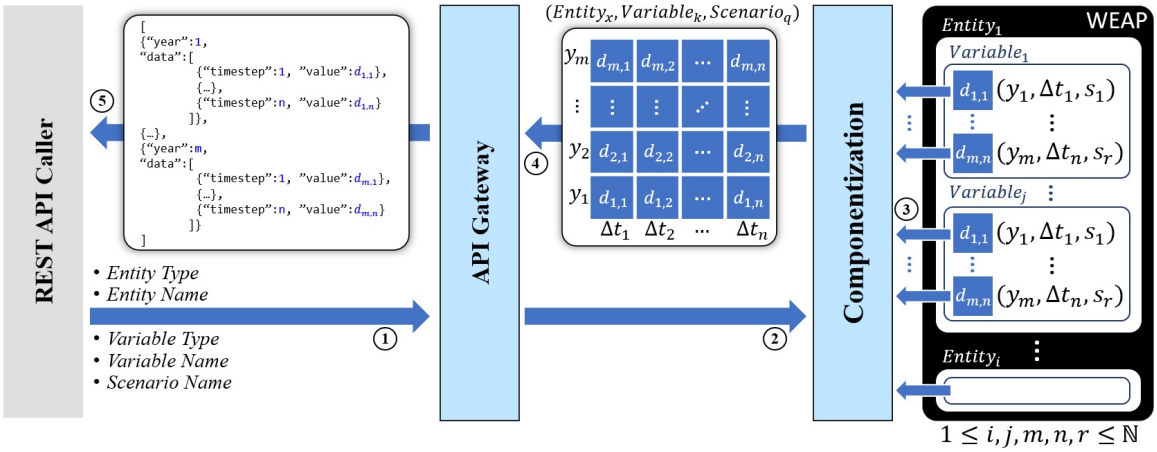


Figure 4.14: Data Retrieving in the Componentized WEAP RESTful Framework.

There are two approaches for the componentization process; *Pre-Componentization* to componentize to entire water model before receiving any request, and *Runtime-Componentization* to componentize the required parts of a model based on the received requests at run-time. The first approach needs considerable time to componentize the whole model, whereas a small part of the model usually interacts with other systems. Furthermore, sometimes the API needs a portion of the values related to a variable (e.g., the first year of values). So, in this research, the Componentized WEAP is developed based on the *Runtime-Componentization* approach. The Componentized WEAP can filter the interval of the data (from $d_{1,1}$ to $d_{m,n}$) for specific years and/or timesteps (using the parameters of the API request). It also has some APIs to control the model execution (i.e., configure, reset, and run the simulation). The frameworks are supported by a local catching mechanism to increase the per-

formance of structure-related requests. It means, given receiving a request related to the structure of a model, the catching system is first checked. In the case of not having the data in the catching system, it will be fetched from the WEAP system and inserted into the catching system for the following requests. The water model can be parameterized and executed, but its model structure cannot be changed using the REST APIs (the water model must be defined inside the WEAP system).

4.1.4 File System

Time-series functions for the variables in the WEAP system do not allow specifying time-step values for all years of the simulation (external CSV/Excel files must be used). Figure 4.15 shows the file system structure used by the Componentized WEAP framework to store the CSV files and use them with the entities in the WEAP system. The Workspace folder is located next to the executable Componentized WEAP file. Two `inputs.csv` and `outputs.csv` files, under the "Project Name" folder, are used to configure the *Min*, *Max*, and *TimeScale* properties for the variables of the WEAP's entities. The data for variables are generated and stored in CSV files under the "Data" folder. The required folder names, such as project name and component type, are retrieved from the invoked URLs. The *ReadFromFile* function (defined in the WEAP system) is used to refer to the CSV files (SEI, 2022d). The folder structure shown in Figure 4.15 prevents any conflict of the data for different projects, components, variables, and scenarios.

4.1.5 Performance Evaluation

Using a MILP Solver (LPSolve, XA, or Gurobi) for the WEAP system, or using any other calculation methods for the entities (e.g., Catchment and Demand Site) do not affect the execution of the entities (i.e., the Componentized WEAP framework

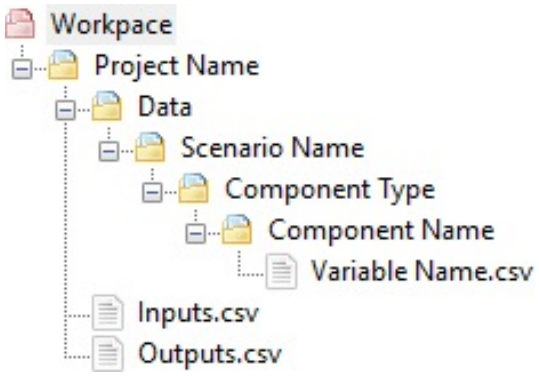


Figure 4.15: The Componentized WEAP Framework File System Structure.

does not affect the execution time for any WEAP model). The execution time of the componentized WEAP is higher than the standalone execution of the WEAP system due to the computation time of the RESTful framework. Changes to any project configuration and scenarios do not change the WEAP entities and their structure (see Figure 4.3).

As described before, the WEAP system is closed-source software. The Automating WEAP scripting language supports the VB-Script, JScript, and Python languages to manipulate and execute WEAP models (SEI, 2022d). The Componentized WEAP system's time efficiency is evaluated against a JScript algorithm (i.e., non-componentized) for the "Weaping River Basin" example (one of the predefined projects in the WEAP system). In both approaches, the simulated experiments have identical set-up (i.e., properties and the values for the input variables are configured). The executions of these simulations are not interrupted and are carried out by the APIs. The Componentized WEAP RESTful framework requires additional steps for identifying WEAP elements, constructing WEAP components, and de/constructing data.

The elements of the "Weaping River Basin" model are 3 *River*, 2 *Reservoir*, 2 *Groundwater*, 6 *Demand Site*, 8 *Transmission Link*, 2 *Wastewater Treatment Plant*, 12 *Return Flow*, 1 *Run of River Hydro*, and 3 *Flow Requirement* entities. The model

is configured for daily and monthly time-steps (12 and 365 steps per year). The efficiency of the Componentized WEAP RESTful framework relative to its proprietary counterpart for each model configuration is compared for a 30-year period (e.g., 2000-2029) with 5-year intervals. Execution times are measured in seconds and averaged over 10 replications. An isolated personal computer with 20 GB RAM and Core i5 Intel CPU on Windows-10 64 bits is used for running all the experiments. The execution times for different time intervals (from 2000 to 2030 with 5 years intervals) in monthly and daily time-steps (12 and 365 steps per year) are collected for the two experimentation settings. All execution times are in seconds and averaged over 10 replications.

Figure 4.16 shows the performance evaluations for the WEAP script and Componentized WEAP RESTful framework simulation experiments. There are slight differences between the componentized and no-componentized models. These differences are due to the available system resources and the creation of the WEAP instance using `ActiveX` and `Winax`. The execution times for the Script and Componentized WEAP RESTful framework are shown in Figure 4.16a and Figure 4.16b for monthly and daily time-steps, respectively. Table 4.3 and Table 4.4 present the minimum, average, maximum, difference, and average data execution times for monthly and daily scenarios. The *Min*, *Max*, and *Dif* data show expected variations in the execution times of the Script and RESTful framework.

Figure 4.16c shows the overhead of using the Componentized WEAP RESTful framework for daily scenarios for a 30-year simulation period with data collected every 5 years. For each year, the times are the time belongs to the componentization. For example, just 10 seconds of 279.9 seconds (see Table 4.4) for a 30-year simulation scenario belongs to the componentization, and the rest is the execution time of the WEAP system. For this configuration, the computation times in Figure 4.16c is the

maximum overhead of the componentization for the "Weaping River Basin" model. The execution times reduce as less data is retrieved (see Figure 9). The execution overhead changes linearly from 0.1 to 10 seconds for a one-year to 30-year simulation. This trend has a direct relation to the number of time-steps per year of the simulation period. For example, the extra computation time for the monthly time-step for the 30-year simulation ($30 \times 12 = 360$ timestamps) is almost the same for the daily time-step for a one-year simulation period ($1 \times 365 = 365$ timestamps).

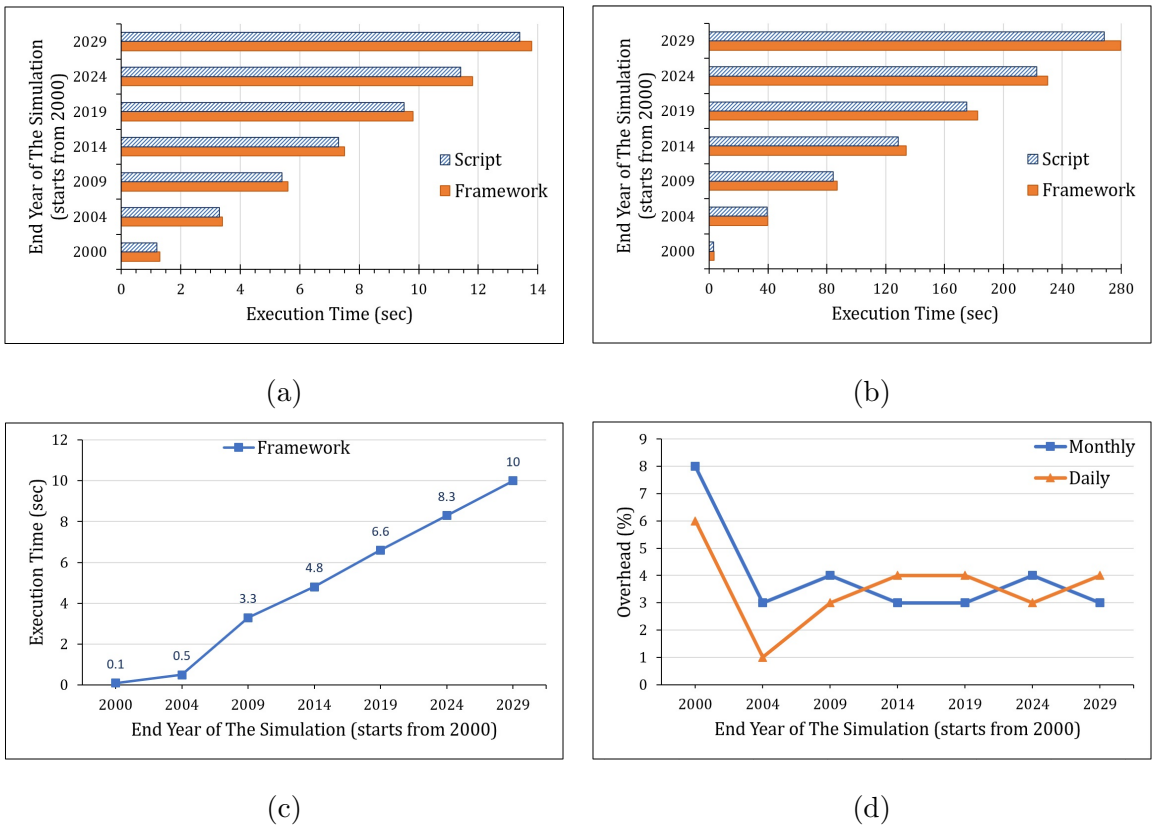


Figure 4.16: The Componentized WEAP RESTful Framework Performance vs. WEAP Script Evaluation. (a) Total Execution Times (Monthly Time-steps). (b) Total Execution Times (Daily Time-steps). (c) Componentized WEAP RESTful Framework Overhead (Daily Time-steps). (d) Componentized WEAP RESTful Framework Overhead in Comparison to the WEAP Script.

The impact of componentizing the WEAP on the total time for simulation studies is negligible. Figure 4.16d presents the overhead percentage in using the Componen-

Table 4.3: The Execution Times (Monthly Time-steps) Using the Componentized WEAP RESTful Framework and the WEAP System Script.

End Year	Script				Framework			
	Min	Max	Dif	Ave	Min	Max	Dif	Ave
2000	1.1	1.3	0.2	1.2	1.2	1.4	0.2	1.3
2004	3.1	3.5	0.4	3.3	3.3	3.6	0.3	3.4
2009	5.3	5.5	0.2	5.4	5.5	5.7	0.2	5.6
2014	7.2	7.3	0.1	7.3	7.4	7.6	0.2	7.5
2019	9.4	9.8	0.4	9.5	9.6	10	0.4	9.8
2024	11.3	11.5	0.2	11.4	11.7	11.9	0.2	11.8
2029	13.2	13.8	0.6	13.4	13.8	14	0.2	13.8

Table 4.4: The Execution Times (Daily Time-steps) Using the Componentized WEAP RESTful Framework and the WEAP System Script.

End Year	Script				Framework			
	Min	Max	Dif	Ave	Min	Max	Dif	Ave
2000	3	3.3	0.3	3.1	3.3	3.4	0.1	3.3
2004	39.1	40.1	1	39.5	39.1	40.2	1.1	39.9
2009	84.2	85	0.8	84.5	87	87.6	0.6	87.2
2014	128.3	130.2	1.9	128.8	133.3	134.1	0.8	133.8
2019	174.1	175.5	1.4	175.3	181.7	184.6	2.9	182.6
2024	222.4	223.5	1.1	222.9	229	230.9	1.9	230
2029	268.4	269.2	0.8	268.7	278	280.4	2.4	279.7

tized WEAP RESTful framework for daily and monthly time-steps (the ratio of the Framework average execution time to the Script average execution time in Table 4.3 and Table 4.4) for the "Weaping River Basin" model. The most considerable overhead at $\sim 8\%$ is for the first simulation period with the daily time-step. This overhead can

be attributed to the model and simulation initialization. For the subsequent simulation periods, the maximum overhead monthly and daily time-steps ranges between 3% and 4%.

The impact of the WEAP system componentization on the simulation execution time is observed to have a direct relation to the number of timestamps, while the scale of a model does not. The overhead of the Componentized WEAP RESTful framework (CWO) function is defined as

$$CWO = CI + CWE + DS \times TS \tag{4.5}$$

where CI is for component identification time, CWE is for Componentized WEAP RESTful framework execution time, and $TS$is the number of timestamps for the duration $((EndYear - StartYear) \times \#TimeStepPerYear)$ of simulation experiment and DS is for the data de/construction time. The CI factor has a generic implementation, so it has a constant value for a model. The CWE and DS factors have constant values for a given model. According to these factors, below a threshold value for the number of timestamps (i.e., executing 400 runs of the "Weaping River Basin" simulation), the CI and CWE play the dominant role in total simulation execution time; otherwise, the TS is the main contributor to total simulation time. For the above simulation experiments, the overhead for the Componentized WEAP RESTful framework executing on a monthly time-step was 0.1 seconds for all time intervals. However, it increased for daily time-steps, as shown in Figure 4.16c. The execution time ratio of the Componentized WEAP RESTful framework over the Script WEAP system is defined as

$$Ratio = \frac{Init + WE + CWO}{Init + WE} \tag{4.6}$$

where the $Init$ is for the time period required for initializing the WEAP system, and the WE is for the time period needed to execute a model in the WEAP system. The $Init$ and WE factors belong to the WEAP system and are directly related to a model's scale. Thus, the ratio of using the Componentized WEAP RESTful framework vs. WEAP script will decrease as the scale of the model increases. Table 4.5 and Table 4.6 present the execution times of a more complex "Weaping River Basin" model (the number of entities is 3-times of the previous experiment) for one-year, 15-year, and 30-year simulation periods for Monthly and Daily timesteps. The tables show in these configurations that the componentization's overhead decreases by increasing the scale of the model, having longer execution time for the WEAP system (WE), and constant Componentized WEAP RESTful framework execution time overhead (CW).

Table 4.5: The Execution Times (Monthly Time-steps) of the Complex "Weaping River Basin" Model Using the Componentized WEAP RESTful Framework and the WEAP Script.

End Year	**Script**				**Framework**				**Ratio**
	Min	**Max**	**Dif**	**Ave**	**Min**	**Max**	**Dif**	**Ave**	
2000	1.9	2.2	0.3	2	1.9	2.4	0.5	2.2	1.1
2014	12.7	14.4	1.7	13.3	13.3	14.7	1.4	13.8	1.04
2029	23.9	25.4	1.5	24.5	24.1	25.7	1.6	24.8	1.01

4.1.6 Framework Software

The Componentized WEAP is a web-service framework that uses the NodeJS (Cantelon et al., 2014) and Typescript frameworks for implementing the server-side application. Typescript is an open-source framework and the superset of JavaScript

Table 4.6: The Execution Times (Daily Time-steps) of the Complex "Weaping River Basin" Model Using the Componentized WEAP RESTful Framework and the WEAP Script.

End Year	Script				Framework				Ratio
	Min	Max	Dif	Ave	Min	Max	Dif	Ave	
2000	5.8	6.6	0.8	6	6	7.1	1.1	6.3	1.05
2014	228.9	238.5	10.4	232.4	228.2	238.5	10.3	232.2	1
2029	455.5	460.3	4.8	458.2	463.4	472	8.6	468.6	1.02

(Wittgenstein, 2012), which has some added and facilitated features (strongly typed programming, module and namespace, generic, interface, and abstraction). The current Componentized WEAP implementation requires using version 2021.0 of the commercial WEAP system (SEI, 2022d). Stockholm Environment Institute (SEI) can publish new versions of the WEAP system, but thus far, the changes are UI-related. The WEAP system has numerous APIs, but the Componentized WEAP framework uses a portion of them (see Appendix A). Changes to the Componentized WEAP RESTful framework are not anticipated, as the models have remained unchanged for several years. Furthermore, making changes to the optimization solvers does not affect the Componentized WEAP RESTful framework. Thus, if no APIs can have side effects on the Entities, Project, and Scenario, they do not cause changes to the RESTful framework. However, changes to the Componentized WEAP framework are expected as the RESTful framework, and its enabling APIs are expected to evolve in the future.

Due to the use of the WEAP APIs listed in Appendix A, it is necessary to have a WEAP system license to use the Componentized WEAP framework. The executable version of the Componentized WEAP framework and a User-Guide are available

(ACIMS, 2022b). The main packages which have been used to develop the Componentized WEAP framework are `TS-Node 8.10.2` (Typescript-Node) to use Typescript in the NodeJS server-side application; `Express 4.17.1` to build a web application and APIs; `Routing-Controller 0.9.0` to create structured, declarative, and beautifully organized class-based controllers; `Body-Parser 1.19.0` to parse the body of the incoming request to web-server, and `Winax 3.1.5` to define `ActiveXObject` in NodeJS (create WEAP instance in server-side application), and some additional packages; for example, `class-transformer`, `class-validator`, and `reflect-metadata`.

Figure 4.17 illustrates the frameworks and tools used in the interaction model and the Componentized WEAP RESTful framework. The number between parentheses for each tool/framework presents the version used. The interaction model is implemented using Java 11 (Java application), and the Jersey (Kalin, 2013) framework is used to build and invoke the APIs of the Componentized WEAP (and Componentized LEAP) RESTful framework. The WEAP and LEAP systems can act as standard "COM Automation Server", and other programming languages (e.g., Visual Basic or C) or scripting languages (e.g., VB Script, JavaScript, Perl, and Python) can interact with the systems via APIs (SEI, 2022d).

The Componentized WEAP is wrapped in a web-service framework implemented in the `NodeJS` and `TypeScript` frameworks. They have been chosen because NodeJS (Cantelon et al., 2014) is an Event-Driven, Non-Blocking I/O Model and Open-Source C++ framework. It is built on V8 (the engine was written by google) and adds some features to handle the JavaScript server-side programming. Also, TypeScript (Wittgenstein, 2012) is an open-source framework and a superset of JavaScript with some added and facilitated features (strongly typed programming, module, namespace, generic, interface, and abstraction).

Popular NodeJS frameworks have been used to handle routine tasks. The *express.js* is a minimal and flexible framework to build web-based architectures. The *routing-controllers.js* and its dependencies (*class-transformer.js* and *class-validator.js*) allow creating controller classes with methods as actions that handle requests. The framework's middleware ("Using Express Middleware", 2022) processes incoming requests before calling the appropriate methods from the controllers of the Componentized WEAP framework. The *winax.js* defines an `ActiveXObject` for the WEAP system in the NodeJS web-server. The architecture of the Componentized WEAP is used for the Componentized LEAP RESTful framework, as well. The current Componentized WEAP and Componentized LEAP implementations require using the commercial WEAP (versions 2021.0) and LEAP (version 2020.1.0.33/32-Bit) systems.

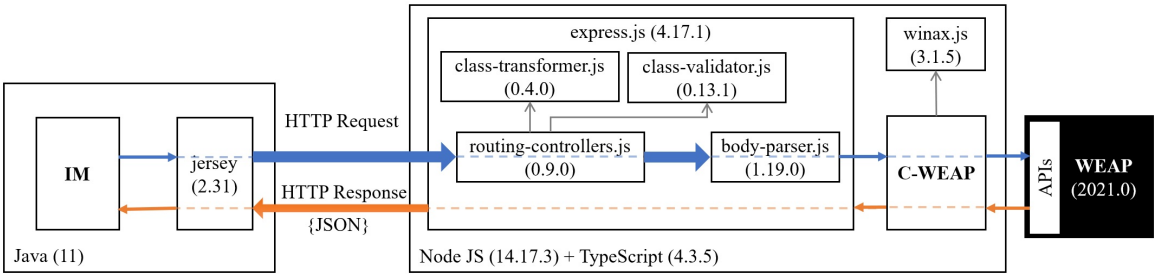


Figure 4.17: The Used Libraries and Packages for the Interaction Model and the Componentized WEAP RESTful Framework.

4.2 Web-Service Framework for the LEAP System

The same scenario applied to the WEAP system is applicable to the LEAP system to use with other simulations. Also, the WEAP and LEAP have many similarities from the structure and behavior point of view. So, the Componentized LEAP is a framework built using the SOA architecture, and the data cube structure (see Figure 2.8) allows integrating it with tools for simulating the water system. According to the constraint for using the JavaScript language and the difficulties of using XML-

based protocols (Tihomirovs & Grabis, 2016), the RESTful framework is used to implement the web-service framework for the LEAP system.

4.2.1 Models of the LEAP Entities

The same steps to componentize the WEAP system are applicable to the LEAP system. So, the Ecore meta-model is used to model the LEAP entities at an abstract view without specifying their functions. At this abstraction level, the data structure of different LEAP sections, entities, variables, and the relationship among these parts are modeled. The specification in Figure 4.18 is defined using the EClass, EAttribute, EDataType, and EReference elements of the Ecore meta-model diagrams. Like the WEAP system, the LEAP's APIs expose the scope and functionality of the componentized entities defined for the LEAP framework.

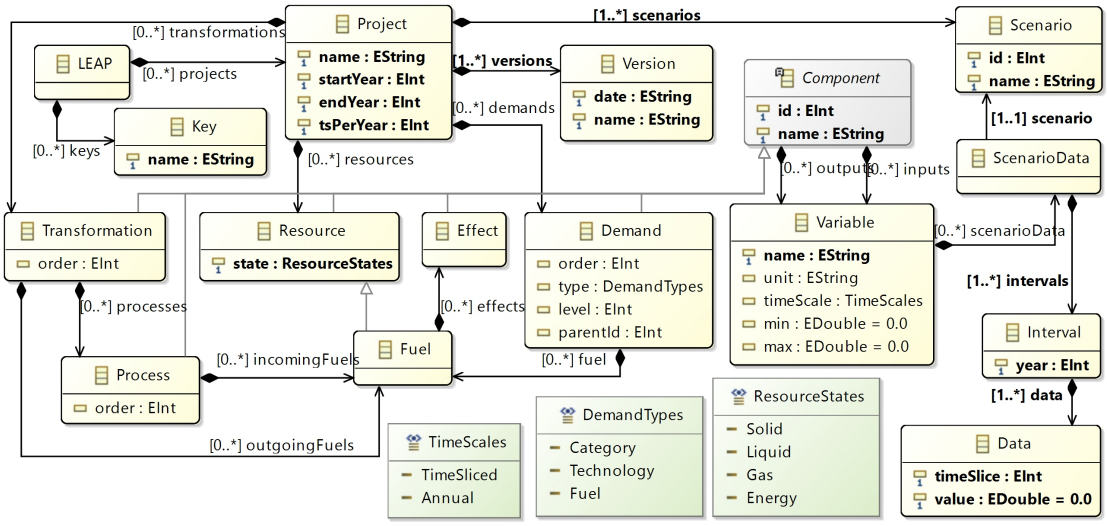


Figure 4.18: Ecore Specification to Model Entities, Variables, and Data of the LEAP System.

In Figure 4.18, the `LEAP` class has an array of projects, each associated with a separate area. Each `Project` has its own configuration (*name*, *startYear*, *endYear*, and *tsPerYear*). The LEAP system has five types of components which are represented

via the `Resource`, `Transformation`, `Demand`, `Process`, and `Effect` classes. The `Component` is an abstract class, and the rest are concrete classes. A `Transformation` entity gets some fuels as the input (*incomingFuels* composite relation between `Transformation` and `Fuel` classes in Figure 4.18), do some process on them (processes composite relation between `Transformation` and `Process` classes in Figure 4.18), and generate some fuels as the output (*outgoingFuels* composite relation between `Transformation` and `Fuel` classes in Figure 4.18). A `Demand` can have multiple fuels to consume. Furthermore, a `Fuel` can have multiple effects (most of the time, the environmental effects).

Like the WEAP system, each entity in the LEAP system has some predefined input and output variables. New variables and equations may be added by users as needed. For each variable, one or more intervals are defined per scenario, and each interval can have many data values (a value represents a specific time-slice of a year). The `Variable` class in Figure 4.18 has a unique *name* property as the key with *unit*, *timeScale*, *min*, and *max* properties. A variable can have just one value per year if the *timeScale* property sets to "Annual", or it can have multiple values if *timeScale* sets to "TimeSliced". The properties *min* and *max* place constraints on the acceptable values for a variable.

A simulation model in the LEAP system has a structure defined by the modeler, but the behaviors for the specialized entities are predefined. The date specified in a scenario is needed to simulate some aspects of an energy system. Like the WEAP system, every project has at least one scenario, the *Current Accounts*, which provides a snapshot of actual energy resources, supplies, and demands. The *name* property is the key attribute in the `Project`, `Scenario`, `Variable`, `Component`, and `Key` classes (see Figure 4.18). The key for the `Version` class is the concatenation of its properties (*date* and *name*). The Componentized LEAP RESTful framework has the

same schema (a generic view) for all the LEAP entity types (e.g., `Demand`, `Resource`, and `Transformation`) and their variables. The `Variable`, `ScenarioData`, `Scenario`, `Interval`, and `Data` classes with their relations define the overall input data and output result for a model in the Componentized LEAP RESTful framework that mirrors those defined in the LEAP system.

4.2.2 Mapping Componentized LEAP Models to a RESTful framework

The Componentized LEAP RESTful framework operates on the LEAP resources (like what was presented for the Componentized WEAP). The communication data is in JSON format. The RESTful API categories for the Componentized LEAP are `Project`, `Version`, `Key`, `Demand`, `Resource`, and `Transformation`. The pattern's structure is like what was presented for the Componentized WEAP. The URL patterns for six API types are shown in Table 4.7. There is a mapping between the URL patterns in Table 4.7 and the Ecore specifications in Figure 4.18. All URLs start with the constant "/Energy", which refers to the `LEAP` class shown in Figure 4.18. The `VariableType` in the URL patterns must be replaced by the "`Inputs`" or "`Outputs`" (refer to the Data or Result variable in the LEAP system). For example, a model (e.g., *demo*) can be executed using the URL "/Energy/demo/Run".

4.2.3 Componentized LEAP Retrieving Data

Provided LEAP APIs do not return the value of a variable in a time-slice granularity. As explained in Section 2.4.2, multiple time-slices (for a year) can be defined in the LEAP system, and they can have different sizes (but their accumulation must cover the whole year). This fine-grain resolution can be used inside the LEAP tool, but it is not accessible via its APIs. However, the individual year values of a Data or Result variable can be stored in the flat file. Figure 4.19 presents the flowchart of

Table 4.7: URL Signatures for Different Types of Componentized LEAP APIs.

Category	URL Signatures
Project	/Energy[/:projectName[/Run]]
Version	/Energy/:projectName/Versions[/:versionName/Revert]
Key	/Energy/:projectName/Keys[/:KeyName/: scenarioName[/Expression]]
Demand	/Energy/:projectName/Demands[/: demandName[/VariableType[/:variableName/: scenarioName[/Expression][?startYear=N&endYear=N]]]]
Resource	/Energy/:projectName/Resources[/: resourceName[/VariableType[/:variableName/: scenarioName[/Expression][?startYear=N&endYear=N]]]]
Transfor- mation	/Energy/:projectName/Transformations[/: transformationName[/Processes[/: processName]][/VariableType[/:variableName/: scenarioName[/Expression][?startYear=N&endYear =N]]]]

retrieving sliced-based data (in the data access layer) in the Componentized LEAP RESTful framework. It starts by checking that the temp directory exists to save the temporal flat files and creates it if it does not exist. Then, in a loop format from the start year to the end year of the simulation, it stores each year's data in a flat file, reads the flat file, and adds the read values to a result object (an array of `Interval` objects in Figure 4.18). Finally, it returns the result object. As can be seen, retrieving variable data in time-sliced resolution is not as trivial as the Componentized WEAP framework. Creating, writing, and reading the flat files for operations to retrieve

the variable's data in a time-slice-based makes the Componentized LEAP framework slower than the corresponding operations in the Componentized WEAP framework.

The design and implementation of the Componentized LEAP framework follow the structures that have been presented for the Componentized WEAP framework (see Section 4.1.2). However, the defined classes for the domains, controllers, services, and data access objects are different (relevant to the LEAP system). Also, the Componentized LEAP framework has the same file system structure as presented for the Componentized WEAP framework to store and use the generated flat files (see Section 4.1.4).

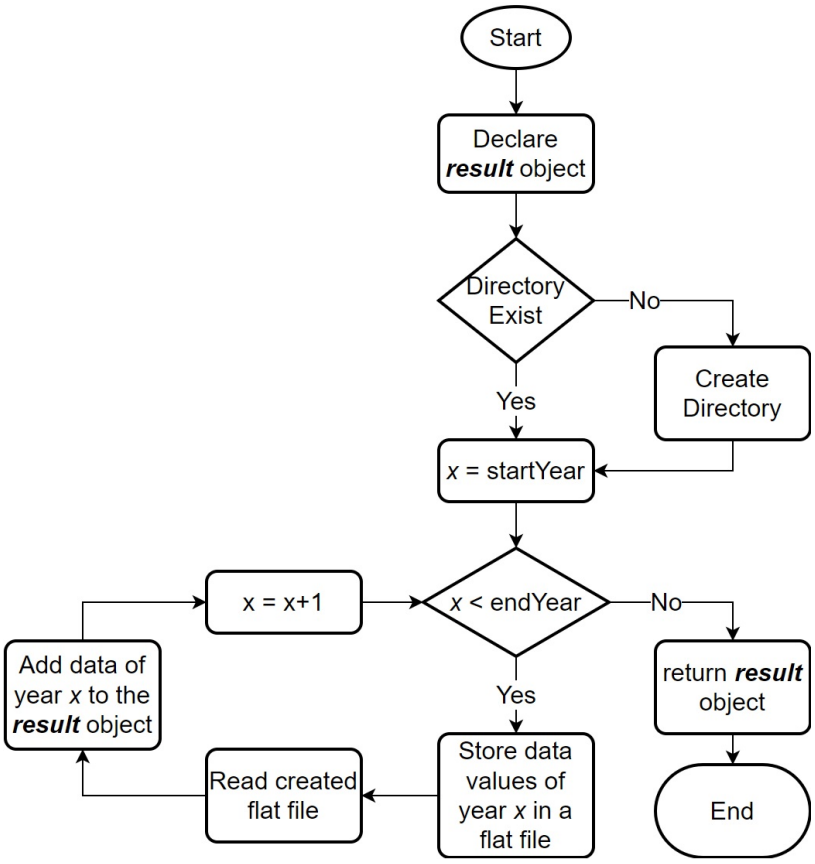


Figure 4.19: Flowchart of Retrieving Sliced-based Data in the LEAP RESTful Framework.

Chapter 5

INTERACTION MODEL DESIGN & IMPLEMENTATION

5.1 Approach

The KIB approach has been introduced to formalize the interactions between the models specified in different modeling formalisms (Sarjoughian, 2006). The KIB formalism can be used to define *data mappings*, *synchronization*, *concurrency*, and *timing*. The conceptual basis of the KIB disparities between different syntaxes and semantics need to be accounted for with a separate model syntax and semantics. Thus, it enables independent modeling of interactions between the composed models. This approach has been applied to different domains (Barton et al., 2016; Huang et al., 2009). In this research, the KIB concept is used to define the relationship between the WEAP and LEAP models externally.

The architecture presented in Figure 5.1 illustrates the abstract specification of the interaction between the WEAP and LEAP systems (using componentized frameworks). Choices of the water and energy model components are defined via the ports between the interaction model and the disparate systems. The constraint under which the data can be transformed for use by one another is defined in the Data Transformation part of the interaction model. The data sets in the water and energy models are used to define a set of modules (data transformation schema) that can be executed under a time-based control regime. An execution protocol prescribes a control regime supported by synchronous calls and returns enabled by RESTful web services and JSON. A simple time management scheme is devised to synchronize different time resolutions used in the water and energy models. The interaction model has interfaces

of the Componentized WEAP and Componentized LEAP frameworks. The WEAP Interface is defined as a part of the interaction model for bi-directional, synchronous communication with the Componentized WEAP models. It also mediates communication between the RESTful web services of the Componentized WEAP and the modules in the interaction model. In response, the Componentized WEAP returns the data (JSON) from the water system to the WEAP Interface. The LEAP Interface is defined in the same way as the WEAP Interface.

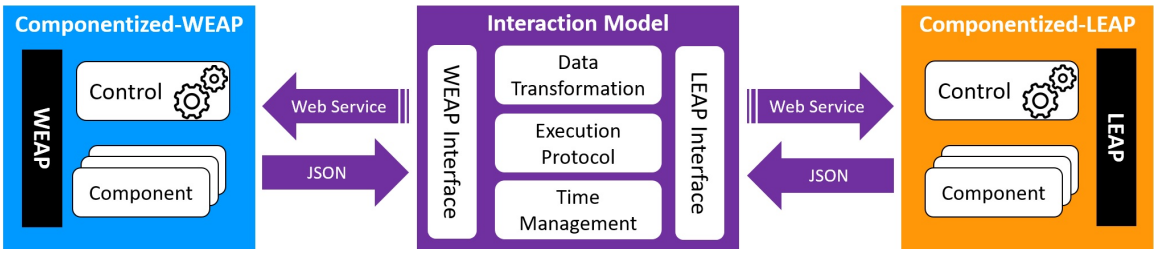


Figure 5.1: A Water-Energy Nexus High-level Architecture in the Algorithmic-IM.

5.2 Algorithmic Interaction Model

As a first attempt to address the need for composing heterogeneous models, this research presents a new modeling and simulation framework based on the Knowledge Interchange Broker (KIB) approach. It helps to understand and assess the FEW-Nexus and coupling developed models in the WEAP and LEAP systems (using the Componentized WEAP and Componentized LEAP frameworks). Knowing how each system interacts with another system is essential to understand them together. Even though WEAP and LEAP tools are internally linked, defining interactions between water and energy models in their internal linkage is limited in terms of flexibly defining choices of data to be communicated, time resolution, and control. The Algorithmic Interaction Model (Algorithmic-IM) is developed based on the KIB approach to compose the water and energy models. It has a cyclic control regime.

5.2.1 Model Specification

The UML class diagrams for the WEN interaction model are included in the core and WEAP-LEAP package diagrams, as shown in Figure 5.2 and Figure 5.3. The `core` package has `IM`, `Coupling`, `TransformationInputPort`, and `TransformationOutputPort` concrete classes (the yellow color classes) that must be instantiated, and the interface `IMessage` (the white color classes) that must be implemented. The `System`, `Module`, `ModuleInputPort`, and `ModuleOutputPort` abstract classes (the gray color classes) must be inherited for a specific coupled Water-Energy model. Following the internally integrated WEAP and LEAP system execution mechanism (SEI, 2022d; Yates et al., 2005), the interaction model executes any two composed water and energy models in a round-based fashion. The `ComponentTypes` in the `Component` abstract class is either the `Module` or `Transformation` enumerated datatype. Also, the *PortTypes* in the `Port` abstract class is either the *Input* or *Output* enumerated datatype. For brevity, the *setter* and *getter* operations are excluded from the diagrams.

The `IM` class has *totalRound* and *currentRound* properties to set the total number of rounds for execution and present the current round of execution, respectively. It also has two *run()* and *run(count)* methods to execute the interaction model. The former method is used to execute the interaction model for *totalRound-currentRound* rounds. The latter method is used to execute the model for *min(totalRound-currentRound, count)* rounds.

In Figure 5.2, the `IM` class has *toCSV()* method to export data to *CSV* file. The `IM` class contains a set of modules (the composite relation from `IM` class to the `Module` class). Each `Module` contains input and output ports (the composite relations from `Module` class to the `ModuleInputPort` and `ModuleOutputPort` classes) as the connection points between the module and the external world. A `Module`

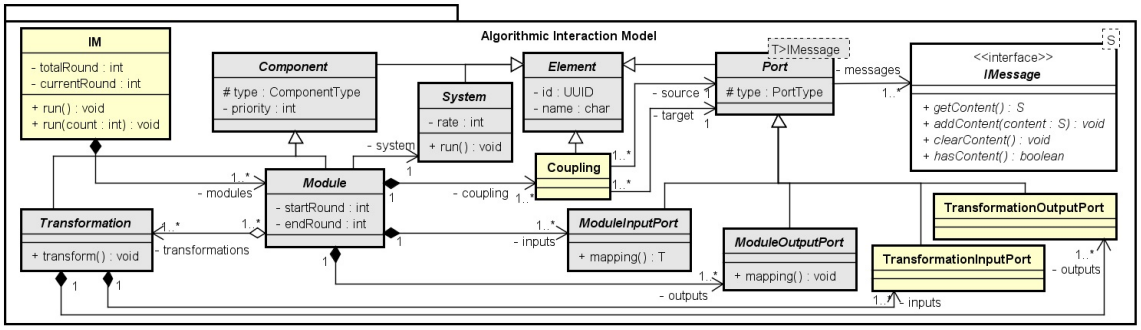


Figure 5.2: A Class Diagram for the Core Package of the Algorithmic-IM.

class also contains a set of transformations (the aggregate relation from the `Module` class to `Transformation` class), and the couplings inside the module. The *isActive(...)* method is used during the simulation to determine if the module is active in a specific round or not (it returns true if it is active). Each transformation has its input and output ports (composite relations from `Transformation` class to the `TransformationInputPort` and `TransformationOutputPort` classes).

The `WEAP` and `LEAP` classes in Figure 5.3 are proxies for the Componentized WEAP and Componentized LEAP models. The module input ports (i.e., `InputFromWEAP` and `InputFromLEAP` classes) and the module output ports (i.e., `OutputToWEAP` and `OutputToLEAP` classes) are defined for the connected ports to the Componentized WEAP and Componentized LEAP models. Few attributes (*projName*, *compType*, *compName*, etc.) are defined to make the APIs for the related systems (attributes for these classes are the same) (Fard & Sarjoughian, 2019). The module ports are entirely independent of each other, and they can have different attributes and operations. The `WEAPMessage` and `LEAPMessage` implement the `IMessage` interface. They are defined as the structures for the incoming/outgoing messages from/to the Componentized WEAP and Componentized LEAP model (see Figure 5.3). Each `WEAPMessage/LEAPMessage` has a finite number of time intervals, subject to the constraints of the frameworks, with each time interval having a time-step/time-slice and

a value (see Section 2.4.3). As an example, two modules (named "Module1" and "Module2") and two transformations (named "Transformation1" and "Transformation2") are defined in the WEAP-LEAP package diagram are instantiated from the `Module` and `Transformation` abstract classes, respectively (see Figure5.3).

The interaction model interacts with external systems using the inputs and outputs defined for its modules. The interaction model defined for the WEN system communicates with the RESTful Componentized WEAP and Componentized LEAP frameworks. The input and output ports for the modules are independent as well as those that are defined for the transformations. All connections within the interaction model are uni-directional. In Figure 5.4, the cloud shape represents the web-server with the APIs defined for the Componentized WEAP and Componentized LEAP models. The APIs are depicted as circles in the cloud. Each component is shown as a rounded rectangle with its input and output ports shown as arrows. Multiple APIs can communicate with a component. Each API can read the input or output data of a component or apply some changes to the inputs. As an example, the URL "/Water/demo/DemandSites/phoenix/Inputs/Annual Activity Level/Reference" accesses the data of the *Annual Activity Level* input variable of the *phoenix* demand site of the *Reference* scenario in the *demo* project of the WEAP system (see Section 4.2.2). The relevant URLs to call APIs from the Componentized WEAP/LEAP models can be defined in the `InputFromWEAP`/`InputFromLEAP` classes (see Figure 5.3). The *component*, *port*, and *scenario* properties are used to specify values to retrieve data from the external systems. The `OutputToWEAP` and `OutputToLEAP` classes are defined for writing data to the WEAP and LEAP systems. The APIs calls are defined in the *mapping()* method of the classes inherited from the `ModuleInputPort` and `ModuleOutputPort` classes (see Figure 5.2 and Figure 5.3).

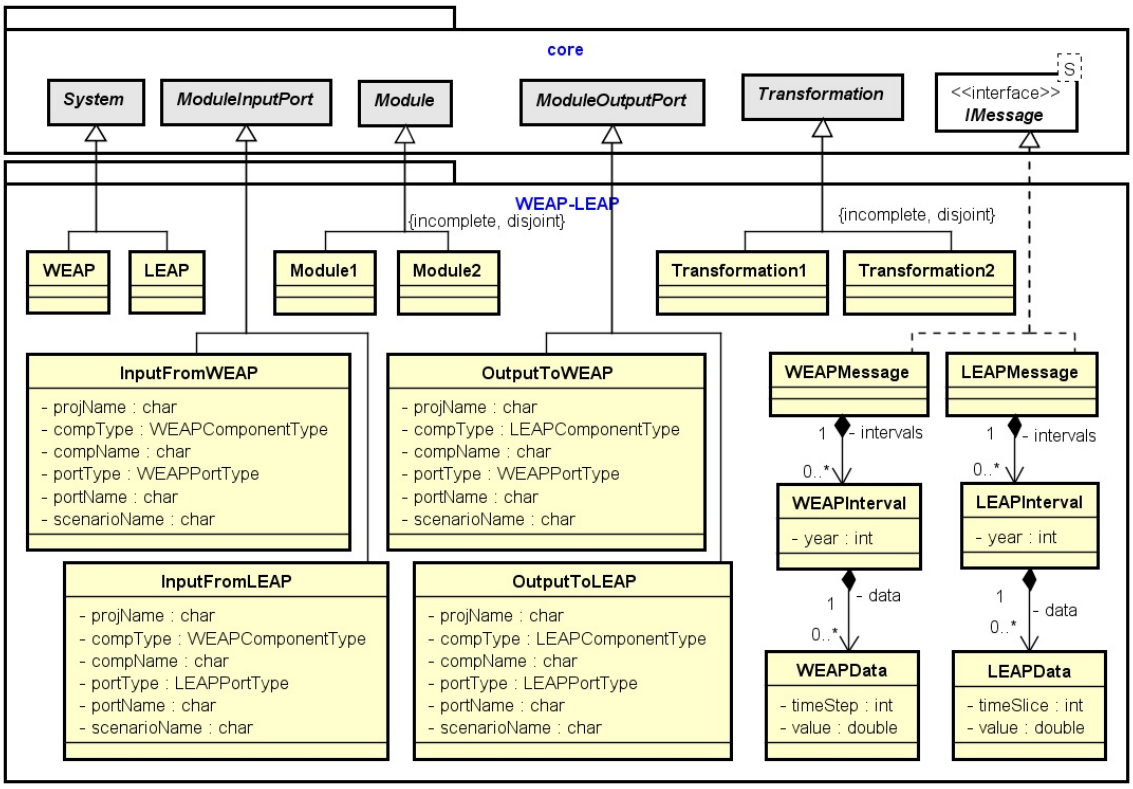


Figure 5.3: A Class Diagram for the WEAP-LEAP Coupling in the Algorithmic-IM.

All data transformation specifications must follow certain constraints. The name of each element in its content must be unique. Examples include the module's name in the Interaction Model content, the module input/output port's name in a module content, the transformation's name in a module content, and the transformation input/output port's name in a transformation content. Also, there are three valid types of coupling (connections between two ports). First, coupling from a module's input port to a transformation's input port. Second, coupling from a transformation's output port to a module's output port. Third, coupling from a transformation's output port to another transformation's input port. Thus, it is not valid to have a self-coupling for any transformation. Figure 5.4 shows the schema of a valid data transformation for a WEN model. This "IM" contains one module (named "Module_1") with two input ports (named "In1" and "In2") and two output ports (named

"Out1" and "Out2"). These ports are linked to the Componentized WEAP and Componentized LEAP models. The module contains two transformations, "Transformation_1" with two input ports (named "In1" and "In2") and two output ports (named "Out1" and "Out2"), and "Transformation_2" with one input port (named "In1") and one output port (named "Out1"). The module has six couplings, three couplings from module input ports to the transformation input ports (i.e., from "Module_1.In1" to "Transformation_1.In1"), two couplings from transformation output ports to the module output ports (i.e., from "Transformation_2.Out1" to "Module_1.Out2"), and one coupling from the transformation output port to the transformation input port (i.e., from "Transformation_1.Out2" to "Transformation_2.In1"). The structure of incoming or outgoing data on a port (the class implemented the `IMessage` class) cannot change during execution. Each module or transformation port can accept a specific message type. As a result, the source and the target of every coupling must have the same message structure (see Figure 5.2).

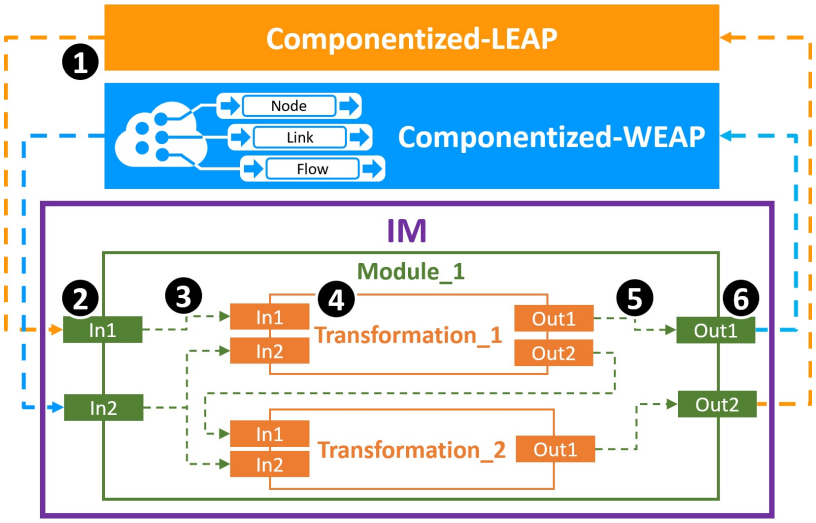


Figure 5.4: An Illustration of the Data Transformation Process for the Coupled Water-Energy System in the Algorithmic-IM.

5.2.2 Execution Protocol

The execution engine for the Algorithmic-IM has a round-based execution. Each complete round consists of six steps, as shown in Algorithm 1. In the first step, the connected systems to the module's input ports are executed. In the second step, the data are invoked by module input ports from the systems. In the third step, the received data are sent to the transformation input ports. In the fourth step, the transformation units are executed (to transform input data to output data). In the fifth step, the processed data are sent to the module's output ports/other transformation input ports. In the sixth step, the output data (collected in the module's output ports) are sent to the systems. Both systems independently and simultaneously execute.

Algorithm 1 presents a pseudo-code for the execution protocol of the Algorithmic-IM. The number of running rounds specifies as the input parameter of the *Run(count)* method. The *totalRound* and *currentRound* properties are referring to the corresponding properties of the `IM` class in Figure 5.2. The S variable, in line 2, presents a set of all connected systems to the interaction model (e.g., the Componentized WEAP and Componentized LEAP systems). In lines 3-5, the *init()* function of all connected systems to the interaction model is executed if it is at the beginning of the simulation execution (current round equals zero). In line 6, the variable *upperBound* sets to the minimum of tr and *currentRound*+*count*, and the body of the while loop (lines 8-41) runs for *upperBound* iterations. As the first step (lines 8-10), the *run()* function of the active systems is executed. A system is active when it is connected to at least one active module and the modulo of the *currentRound* to its rate equals to zero (line 9). In lines 11-14, the active modules are selected using their *isActive(...)* method. In the second step (lines 15-18), the *mapping()* function of the module's

input ports of active modules which are connected to the active systems is executed. In the third step (lines 19-25), the data in the module's input ports (if there is data) are transferred via couplings to the transformations input ports. If a module input port is coupled to multiple transformation's input ports (e.g., module input port "In 2" in Figure 5.4), the data is copied to all destinations (lines 22-24), then the module's input port is vacated (line 25). In the fourth step (lines 26-28), the *transform()* function of all active modules is performed (i.e., all transformation's input data are processed and the results are generated on the transformations' output ports). In the fifth step (lines 29-36), the data on the transformations' output ports are transferred via couplings to the transformations' input ports/module's output ports. Again, if a transformation output port is coupled to multiple destination ports, the data is copied to all the receiving transformations in the module (lines 33-35) and then the transformations' output ports are vacated (line 36). The *currentRound* value is increased by one unit in line 37. In the sixth step (lines 38-41), the mapping() function of the module's output ports of active modules that are connected to the active systems is executed to send data to the systems. Finally, after executing the interaction model for *count* round, all external systems are run to generate their results. It is noted that all modules execute independently of one another. For brevity, the sorting operation for modules, transformations, and ports is not shown in the Algorithm. For example, modules are sorted by priority in line 14, and module input ports are sorted by priority in line 16.

Algorithm 1 Run Simulation

```
1: procedure RUN(count)
2:     S ← distinct Systems connected to the input & output ports of all modules in the IM
3:     if (currentRound == 0) then
4:         for all (s in S) do
5:             s.init()
6:     upperBound ← MIN(count+currentRound, totalRound)
7:     while (currentRound < upperBound) do
8:         for all (s in S) do
9:             if (currentRound % p.system.rate == 0) then
10:                p.system.run()
11:        activeModules ← φ
12:        for all (m in IM.modules) do
13:            if (m.isActive(currentRound)) then
14:                activeModules.add(m)
15:        for all (m in activeModules) do
16:            for all (p in m.inputs) do
17:                if (currentRound % p.system.rate == 0) then
18:                    p.mapping()
19:        for all (m in activeModules) do
20:            for all (p in m.inputs) do
21:                if (p.messages.size() > 0) then
22:                    for all (c in m.couplings) do
23:                        if (c.source == p) then
24:                            c.target.messages ← c.source.messages
25:                    p.messages ← φ
```

Algorithm 1 Continued Run Simulation

```
26:         for all (m in activeModules) do
27:             for all (t in m.transformations) do
28:                 t.transform()
29:         for all (m in activeModules) do
30:             for all (t in m.transformations) do
31:                 for all (p in t.outputs) do
32:                     if (p.messages.size() > 0) then
33:                         for all (c in m.couplings) do
34:                             if (c.source == p) then
35:                                 c.target.messages ← c.source.messages
36:                         p.messages ← φ
37:         currentRound ← currentRound + 1
38:         for all (m in activeModules) do
39:             for all (p in m.outputs) do
40:                 if ((currentRound % p.system.rate == 0) & (p.messages.size() > 0)) then
41:                     p.mapping()
42:     for all (s in S) do
43:         s.run()
```

5.2.3 Time Management

The time resolution for coupled WEAP and LEAP models uses the timing specification defined in the Water-Energy nexus model. The *rate* property in the `System` class (see Figure 5.2) and the execution protocol (see Section 5.2.2) synchronize the

WEAP and LEAP models. The classes implementing the `IMessage` interface and the defined transformations (see Figure 5.2) control matching the time intervals defined for the water and energy models in each execution round. The *rate* property of the `System` class in Figure 5.2 defines the ratio of running the system related to the round of the interaction model. For example, if the time interval for a water model is a year (e.g., the start year is 2020 and the end year is 2021) and the time interval for an energy model is ten years (e.g., the start year is 2020 and the end year is 2029), then the rate of a WEAP must be one, and the rate of the LEAP must be 10 in the interaction model. For a coupled WEAP-LEAP model, the period for the WEAP is given for the first year with all subsequent years to be simulated. In this scenario, for every 10 execution cycles of the WEAP model, there is 1 execution cycle of the LEAP model.

Finer-grain time resolution can be defined using the classes implementing the `IMessage` interface. For example, a WEAP or LEAP model can have a finite discrete-time resolution for a year (Fard & Sarjoughian, 2021b). The time resolutions for the data transformations from the WEAP to the LEAP (or vice versa) follow the restrictions provided in the WEAP and LEAP systems, individually and together. The time-values relevant to the classes realized for the `IMessage` interface (the `WEAPMessage` and `LEAPMessage` in Figure 5.3) can be mapped to some other values by aggregating and/or disaggregating data in the water and energy models. The modeler can define time interval conversions for each data transformation consistent with the time intervals used in the water and energy models.

5.2.4 Algorithmic-IM Configuration

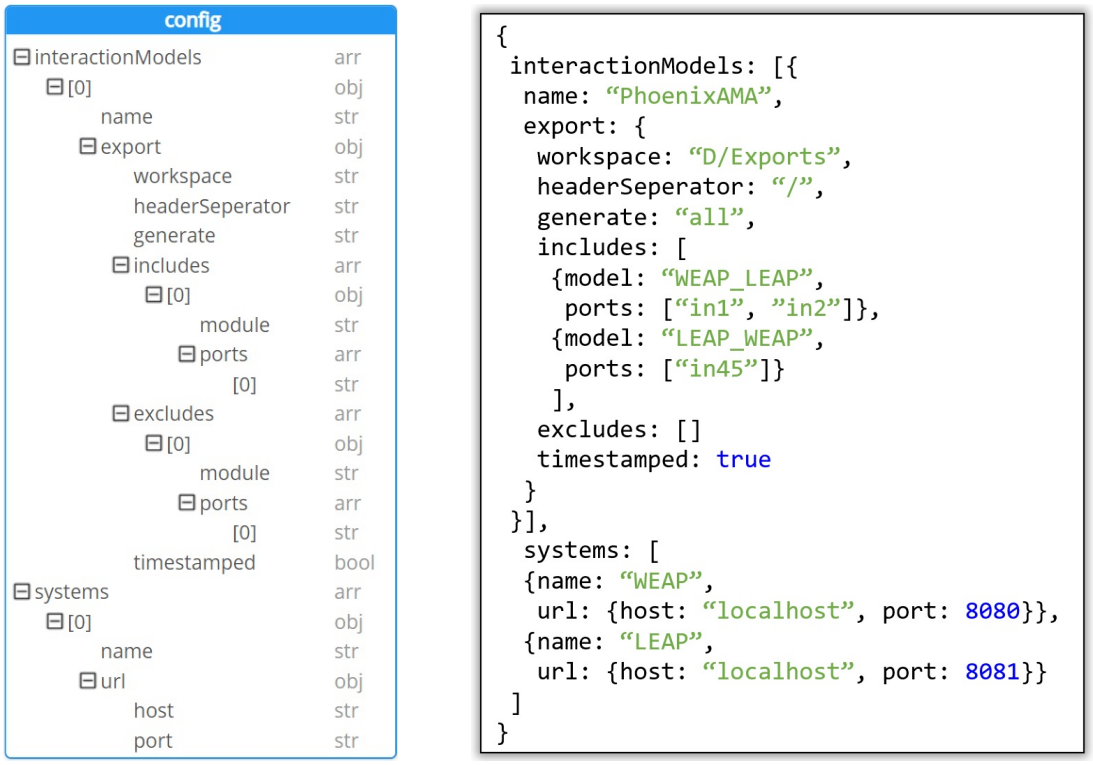
A JSON file, named "config.json", is used to define the configuration for the external systems (connected to the interaction model) and the interaction model.

Figure 5.5a shows the schema of the `config` file, which contains two objects, *interactionModels*, and *systems*. The former is an array of objects for interaction models. Each object has a name (same as the interaction model name) and another object to configure the export to *CSV* file feature (it will be explained in the next section). A system is an array object that contains the connection properties of the external systems. For example, the host and port of the Componentized WEAP and Componentized LEAP frameworks in this research. As an example, Figure 5.5b presents a configuration file for the Algorithmic-IM model defined between the WEAP and LEAP systems (using the Componentized versions) for the Phoenix Active Management Area. The interaction model name is "PhoenixAMA", and it has a configuration for the export feature. The Componentized frameworks are running on the *localhost* and on *8080* and *8081* ports, respectively. So, the interaction model application uses these values at run-time to call Componentized WEAP and Componentized LEAP APIs.

5.3 DEVS-Based Interaction Model

In earlier work, the Algorithmic-IM was proposed and developed to integrate the componentized WEAP and LEAP RESTful frameworks for modeling and simulating water and energy systems. However, this approach does not separate modeling and simulation protocols from each other. It also does not support flexible, structured model hierarchies. To overcome the Algorithmic-IM limitations, the parallel DEVS formalism is used to develop an interaction model with the DEVS-Suite simulator. It is advantageous to use a formal modeling method instead of an algorithm to model and simulate the interactions between the nexus of the water-energy system.

Using a formal modeling method to model and simulate the interactions between disparate models is advantageous. A component-based, hierarchical modeling ap-



(a) (b)

Figure 5.5: Algorithmic-IM Configuration File. (a) Schema. (b) The Phoenix AMA Configuration File as an Example.

proach that aligns with system thinking helps with the interaction model's development, reuse, and maintainability. An interaction model framework is designed and developed based on the KIB approach and DEVS formalism (called the DEVS-IM framework). The DEVS-IM is grounded in system theory and component-based modeling. It has a unified concept for specifying general-purpose logical and persistent atomic and coupled DEVS models. The models are used to specify the hierarchical tree structure of the interaction model. The leaves of the tree structure are atomic DEVS models, and the rest are coupled DEVS models. The input and output ports can be defined for the models as the interface for message communication. The atomic and/or coupled models can be connected using coupling between their ports. The DEVS-IM framework supports storing models in the MongoDB database. Model

creation, access, and manipulation are accessible via REST APIs. The DEVS-IM framework has some predefined elements (derived from the atomic and coupled DEVS models) to facilitate defining the interaction model for the users who are unacquainted with the DEVS formalism. Furthermore, the framework has a set of elements to define a generic ontology for the disparate external systems connected to the interaction model.

The Parallel DEVS formalism (Chow & Zeigler, 1994) is selected for designing the Interaction Model due to its strong modularity, hierarchy, and support for discrete-time state transitions with inputs and outputs used in Componentized WEAP and Componentized LEAP frameworks. Furthermore, it is important to use established modeling and simulation engines. The parallel DEVS models can be developed, simulated, tested, and debugged using the DEVS-Suite simulator (ACIMS, 2022d; McLaughlin & Sarjoughian, 2020). Together, the DEVS formalism and the DEVS-Suite simulator provide a solid advancement to the interaction model's algorithmic approach and implementation.

Figure 5.6 illustrates the conceptual architecture of the DEVS-IM framework. It is divided into three main sections; "IM Model", "External System Schema", and "External System". The first two sections (the purple area) define the interaction model between disparate systems. The "IM Model" section is a coupled DEVS model to realize the KIB properties. The "External System Schema" section defines a tree structure as an interface for the external system. It implements the actual communication between the interaction model and the external systems. From the interaction model perspective, any type of system/model can be considered as an external system; A stand-alone application, web service, database, library, and file system; For example, the developed WEN model in this article (using the WEAP and LEAP sys-

tems) is based on web service communication. The external system schemas can be used to model the external system at different levels of abstraction.

In Figure 5.6, the "Data Transformations" part contains all elements to process the input messages (e.g., aggregate, disaggregate, or convert the data) and then generate output messages. This part is the realization of the *data transformation* property of the KIB approach (the same functionality of the Transformation element in the Algorithmic-IM framework). The data carried in a message in the DEVS-IM framework can be as simple as a primitive data type value (e.g., integer or string) or as complex as a user-defined model. The "Execution Control" part in Figure 5.6 involves *timing*, *synchronization*, and *concurrency* properties of the KIB approach. Each disparate model executes in its framework and follows its execution protocol. As the coordinator, the DEVS-IM model has its execution protocol to coordinate and synchronize the executions among the DEVS-IM models and disparate external systems. Synchronization control is crucial to ensure message ordering and causality among the models. The "Execution Control" section can send control messages (dotted red arrows) to the data transformations or output connectors based on the received data messages (solid red arrows) from input connectors and data transformations. Given different DEVS-IM execution algorithms, the disparate models and the interaction model can be executed sequentially or in parallel. The *Time* property of the KIB in this research focuses on controlling the logical time of the DEVS-IM model and the external systems. It ensures that when a model receives a message from another model (probably with a different modeling formalism), the essential time associated with the message represents the same time in the model. It must be done at the modeling specification and simulation/execution levels.

The "Input Connectors", "Data Transformations", and "Execution Control" in Figure 5.6 are pure DEVS models. The "Output Connectors" communicate with

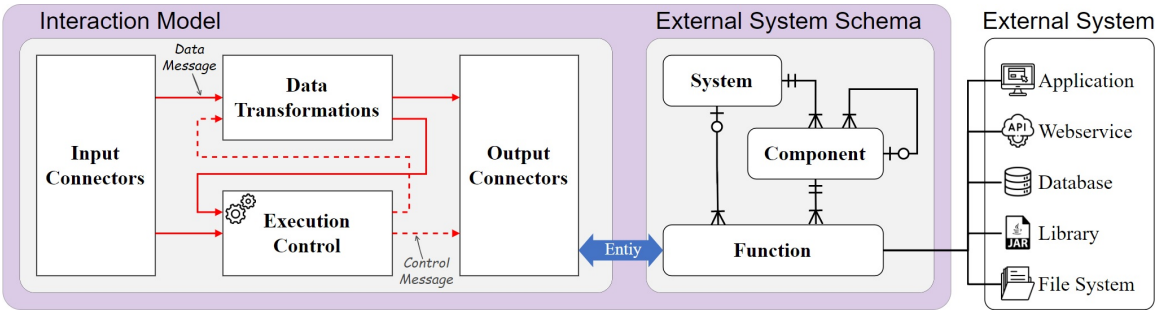


Figure 5.6: Conceptual Architecture of the DEVS-Based Interaction Model Framework.

the outside world via the "External System Schema". A communication message between the "Interaction Model" section (specifically, the "Output Connectors" part) and the "External System Schema" section (specifically, the "Function" part) must be inherited from the `Entity` class (a base class in the DEVS-Suite simulator). So, the "Function" part is responsible for receiving/sending data from/to the external systems in any format or structure. Indeed, the results from the outside world (received in the "Function" part) must be converted to the *Entity* type before returning them to the "Output Connectors" part. During the simulation execution, the received data in the "Output Connectors" part (from the external systems) are sent to the "Input Connectors" (indicated by the solid red arrow from "Output Connectors" to the "Input Connectors" in Figure 5.6). In general, disparate systems present distinct specifications on the model structure. Therefore, the structural composition specification of the KIB is desired to handle the differences in the interface structures between the models in different systems.

5.3.1 Model Specification

Figure 5.7 presents the main steps to model and simulate an interaction model using the DEVS-IM framework. First, the structure of the interaction model must be defined. Second, code is generated for the skeleton of a complete project in the

DEVS-Suite simulator. Third, the behavior of the interaction model is defined by identifying the external connectors and the data transformations under sequential and synchronous control schemes. Forth, the DEVS-Suite simulator is used to test, debug, and run the interaction model.

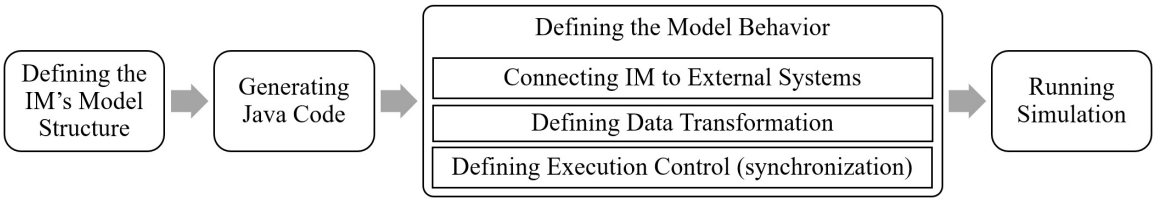


Figure 5.7: Steps of Developing a Model in the DEVS-IM Framework.

The DEVS-IM model can be defined using predefined *IM*, *Input Connector*, *Output Connector*, *Process*, *Task*, *Port*, and *Coupling* elements. It also supports defining an interface for external systems using predefined *System*, *Component*, and *Function* elements. Figure 5.8 illustrates the order of defining different elements of a DEVS-IM model. A *Project* element can contain multiple interaction models between disparate systems. The purple and yellow color elements in Figure 5.8 are used to define the "IM Model" and "External System Schema" sections in Figure 5.6. The *IM* element defines the interaction model between the systems (the root element of an interaction model). It can contain four types of sub-elements/children; *Input Connector*, *Output Connector*, *Process*, and *Task*. The *Input/Output Connector* element defines an interface for the *IM* element to receive/send data from/to the external system schemas. The *Task* element defines a data transformation, and the *Process* element provides the hierarchy modeling. A *Process* element has a set of *Process* and *Task* elements as its sub-elements. The *Port* element (input and output) defines the communication part between *Task*, *Process*, and *Connector* elements. Finally, a Coupling element can be defined between two Port elements or between one *Connector* element and one *Port* element (the solid and dotted red connections in the "IM Model" section in

Figure 5.6). In defining the "External System Schema", the *System* element defines the root node of a tree (usually with the same name as the external system). Multiple *Component* and/or *Function* elements can be defined under the *System* element (see the Entity Relation diagram in the "External System Schema" section in Figure 5.6). Each *Component* element is a representation of a specific entity in the external system. It can contain multiple *Component* and/or *Function* elements to define the hierarchical structure of the tree. The *Function* element (which is always a leaf of the tree) handles a specific functionality of the representative entity in the external system. Each *Function* element has two responsibilities; 1) sending/receiving data to/from the external systems (e.g., calling APIs in the WEN Example), 2) data type conversion between the acceptable type by the DEVS-IM simulator (i.e., Entity class) and an acceptable type/format by the external system (convert Message to JSON and vice versa).

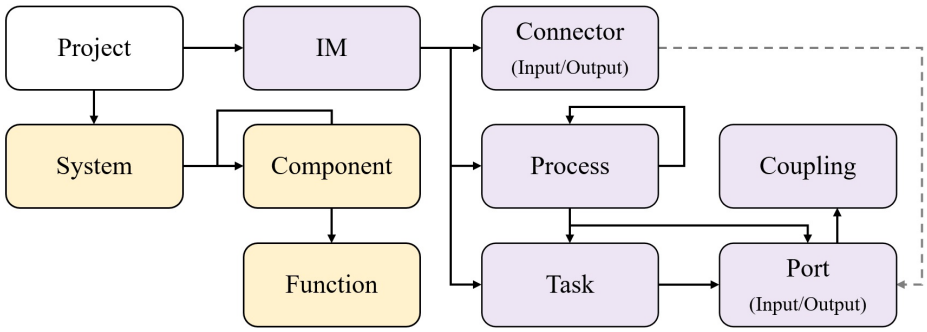


Figure 5.8: Order of Element's Creation in the DEVS-IM Model.

The specification of the interaction model using the parallel DEVS formalism leads to rigorous, systematic model development. This interaction model satisfies two needs. One is to model the interactions among the water and entity models. Another is for the highest level DEVS coupled model to communicate with other simulators. In Figure 5.9, the modeling package highlights the DEVS's core modeling engine. The abstract, interface and concrete classes are shown in gray, white, and yellow colors,

respectively. All classes are inherited from `Entity` abstract class. Some attributes (*id*, *createDate*, *lastModifyDate*, and *version*) will be filled automatically at runtime by the engine (at the time of storing the data in the database). According to the DEVS formalism, there are two main classes to define the models (which are inherited from the `Model` abstract class); the `AtomicModel` and `CoupledModel` classes. The atomic models define the behavior of a system, and the coupled models define the structure of a system. A model has some ports (see the composition relation between the `Model` and `Port` classes). The DEVS formalism just have input and output ports (presented via `InputPort` and `OutputPort` concrete classes in Figure 5.9). Defining the Port class as a high abstract class provides the flexibility to define different types of ports for the interaction models. Like the Algorithmic-IM approach, a port has the messages aggregate relation to the `IMessage` interface. A `CoupledModel` element can have many sub-models (the aggregate relation from the `CoupledModel` class to the Model class) and the couplings between them (the composite relation from the `CoupledModel` class to the `Coupling` class). The `Relation` abstract class is defined for the same purpose described for the `Port` class (having flexibility for future needs). The DEVS formalism supports one type of coupling (which is defined via the `Coupling` concrete class in Figure 5.9). A coupling starts from a port and ends at another port. Based on the DEVS formalism, just valid couplings are allowed to be defined; *External-Input-Coupling* (EIC), *External-Output-Coupling* (EOC), and *Internal-Coupling* (IC).

Figure 5.10 illustrates the class diagram of the `component` package in the DEVS-IM framework. Two main classes in the component package are `IMAtomicModel` and `IMCoupledModel`. The *componentType* attribute in these two classes is an enumeration and can have one "IM", "INPUT_CONNECTOR", "OUTPUT_CONNECTOR", "PROCESS", "TASK", and "LOGIC" values. Each atomic or coupled inherited class sets the *componentType* value in its constructor. The required functionalities

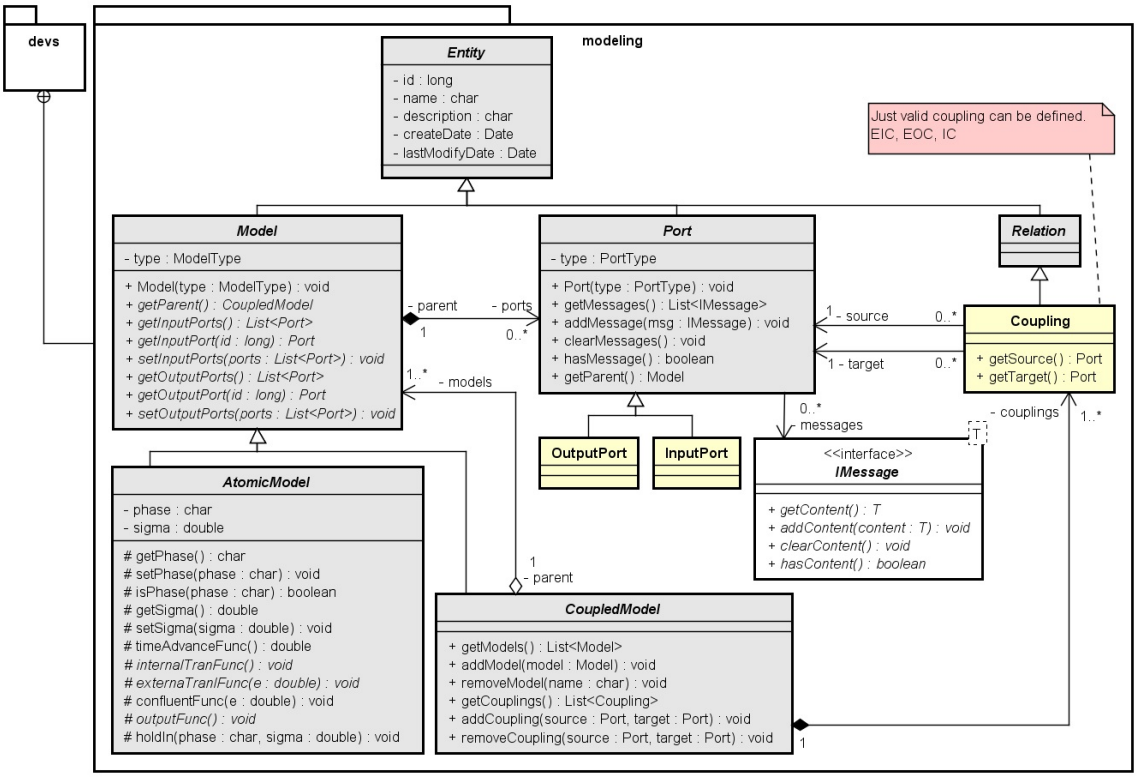


Figure 5.9: The Class Diagram of the Modeling Package for the DEVS-IM Framework.

for the components are defined in their corresponding interfaces. Depending on its functionality, an `OutputConnector` class may select an `InputConnector` class (the input association relation between the classes), depending on its functionality. The `Project` class, in the `im` package, is inherited from the `Entity` class and can have multiple interaction models (see the composition relation between the `Project` and `IM` classes). The `Project`, `IM`, `Process`, and `Task` are concrete classes, and the `Logic`, `InputConnector`, and `OutputConnector` abstract classes must be specialized.

The composition of the WEAP and LEAP models can have one or more interaction model components for a given project. Because the `IM`/`Process` class inherits from the `IMCoupledModel` class, the *IM*/*Process* element can have many *Process*, *Task*, and *Logic* elements as its sub-elements. Each *IM* element can have multiple *InputConnector* and *OutputConnector* elements. These input and output elements,

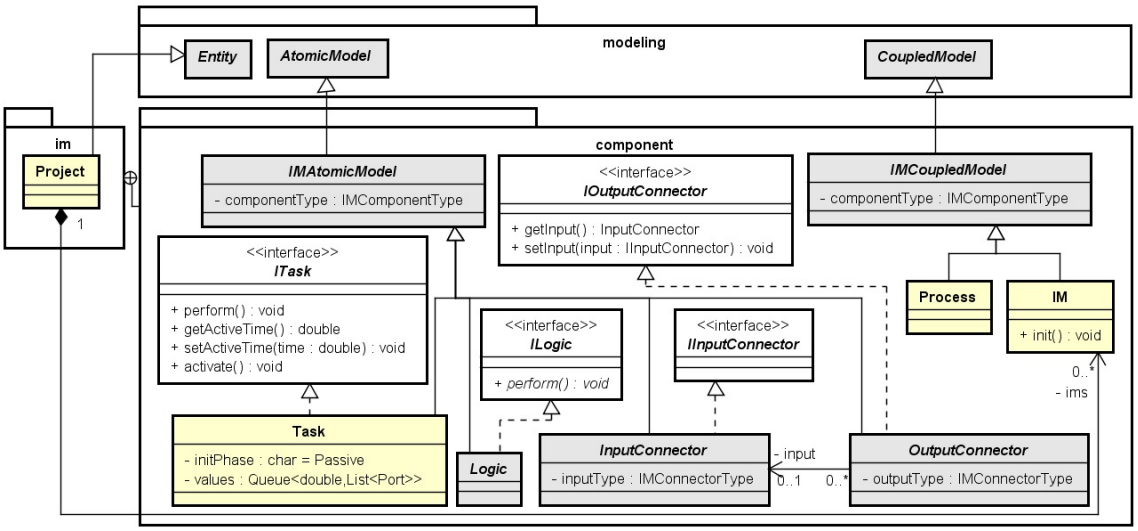


Figure 5.10: The Class Diagram of the Component Package for the DEVS-IM Framework.

defined using DEVS atomic models, can be coupled with any DEVS model and communicate (via function calls) with the Componentized WEAP and Componentized LEAP frameworks. The *Process* element cannot have any *InputConnector* or *OutputConnector* elements as its sub-elements. The DEVS coupled *IM* element does not have any input and output ports. The *init()* method in the `IM` class defines the initialization for the interaction model.

The `system` package in Figure 5.11 includes three concrete classes to represent the external systems (i.e., the WEAP and LEAP tools) in a hierarchical component-based manner. Multiple systems can be defined in each project (see the composite relation from the `Project` class to the `System` class in Figure 5.11). All classes have the *id* (as the key attribute) and unique *name* attributes. The `ISystem` and `IFunction` interfaces defined the method signatures to be implemented in the `System` and `Function` concrete classes. The *init()* and *run()* methods in the `ISystem` are used for initialization (running at the beginning of the interaction model simulation execution) and executing the external simulation system, respectively. A hierarchy of

components can be defined in a system, and each component can have many functions. A `Function` instance can have one parameter of type Object. In the case of having multiple input parameters for a function, they must be wrapped in an object/class. The desired objective of the `Function` must be implemented in the *exec(...)* method. The *getResult()* method returns an `IMessage` as the result of the execution. In the WEAP and LEAP systems, the functions of each component must call the RESTful APIs defined by the Componentized WEAP and Componentized LEAP frameworks. In the Algorithmic-IM framework, all these steps are generalized and handled inside the *mapping()* methods of the module's ports (see Figure 5.2). The Algorithmic-IM did not have any consideration for the details of the external system's models. In the DEVS-IM framework, an interface of the external system must be defined in the interaction model.

As mentioned before, the interaction model connectors are atomic models from the DEVS viewpoint. Simultaneously, they are connectors to the external systems from the interaction model standpoint. Some predefined input and output connectors (with specific behavior) are defined in the current DEVS-IM design (the `predefinedComponent` package in Figure 5.11). The `InputConnector` and `OutputConnector` abstract classes in Figure 5.11 (the DEVS-IM design) have the same role as the `ModuleInputPort` and `ModuleOutputPort` abstract classes in Figure 5.2 (the Algorithmic-IM design). In the module's ports of the Algorithmic-IM, the *mapping()* functions defined the port behavior. However, in the DEVS-IM design, the behavior defines using the DEVS specification functions. From the DEVS specification, all the interaction model connectors have one input port (named "in") and can have one output port (named "out").

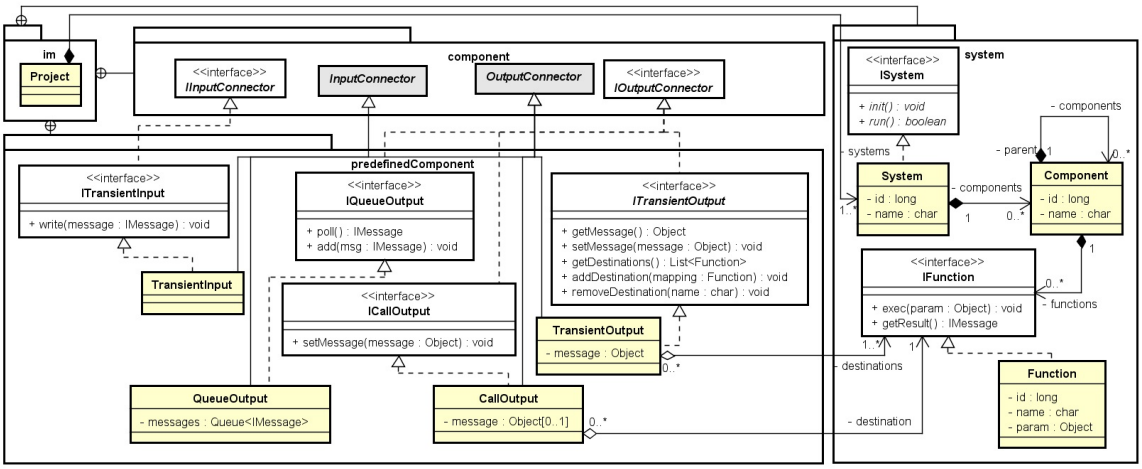


Figure 5.11: The Class Diagram of the Predefined and System Packages for the DEVS-IM Framework.

5.3.2 RESTful Framework Specification

Figure 5.12 illustrates the architecture of manipulating the structure of a model in the DEVS-IM framework. The DEVS-IM is supported by a RESTful framework to define the model's structure and store them in the MongoDB database. Different elements can be created, read, updated, and deleted (CRUD operations) via the HTTP POST, GET, PUT, and DELETE methods. The "REST Request Handler" receives a request and routes it to a proper service. A separate service for each DEVS-IM element is defined in the "Service Handler" section. At this step, the required verifications are checked on the incoming request (from the client), then the changes are applied to the database if all verifications are passed. A proper error message will be returned to "API Caller" if the request is not defined correctly or violates a rule. Otherwise, the database will be changed, and the retrieved data from the database (Domain models) is converted to the Data Transfer Object (DTO) model by the "Service Handler", and then converted to the JSON object by the "REST Request Handler". Eventually, the client receives a response in JSON format.

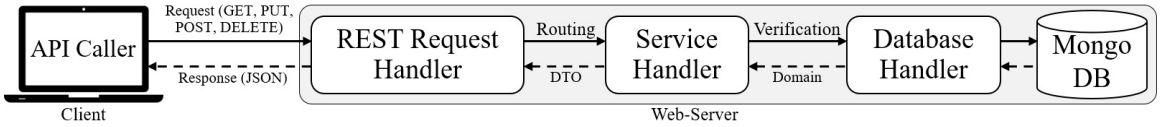


Figure 5.12: The DEVS-IM Dramework Architecture to Define the Model's Structure.

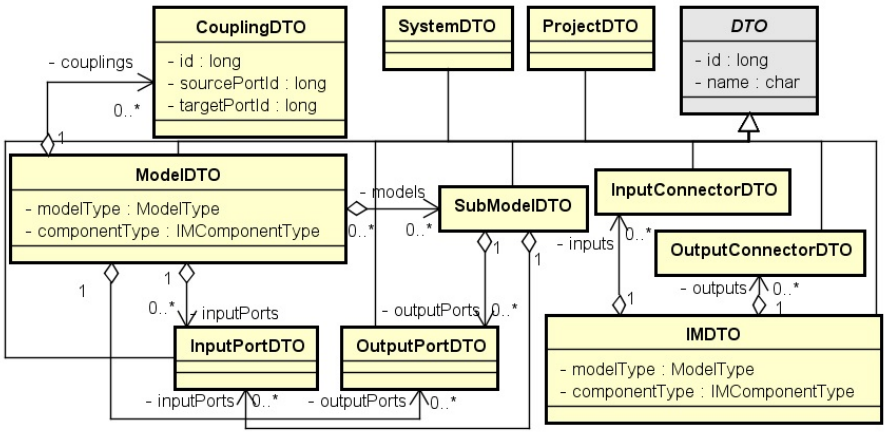
Table 5.1 presents the URL signatures for existing REST APIs of the DEVS-IM framework to define a model. In the signature of the URLs, constants are written in PascalCase style; parameters are written in camelCase style, and they must be inside the open and close pair curly brackets. The content inside each open and closed pair of square brackets is optional to define different APIs for an element. Figure 7 in [54] presents the REST APIs' communication data schema (body of the requests). As an example of using APIs, calling the URL "/IM/Projects" via a POST method with "{"name": "WEN"}" as the body of the request will create a new project and return the Id of the inserted data into the database if it passes the verification phase (i.e. if the project name has a correct pattern and it is not repetitive). As another example, calling the URL "/IM/Models/1" as a GET method will return a list of all sub-models and the couplings of the sub-model with modeled equally to one (given having a valid Id).

In Figure 5.12, the body of the insert/update requests and the communications between "REST Request Handler" and "Service Handler" sections are based on the Data Transfer Objects (DTOs). Figure 5.13a and Figure 5.13b present the DTOs to retrieve data (the response of the GET requests) and insert/update data (the body of the POST or PUT requests), respectively. The delete operation handles by setting the *id* of an element as a URL parameter. The data needed for the RESTful framework is in JSON format. The {JsonIgnore} constraint for some attributes in Figure 48(b) indicates that these attributes are hidden for the "API Caller".

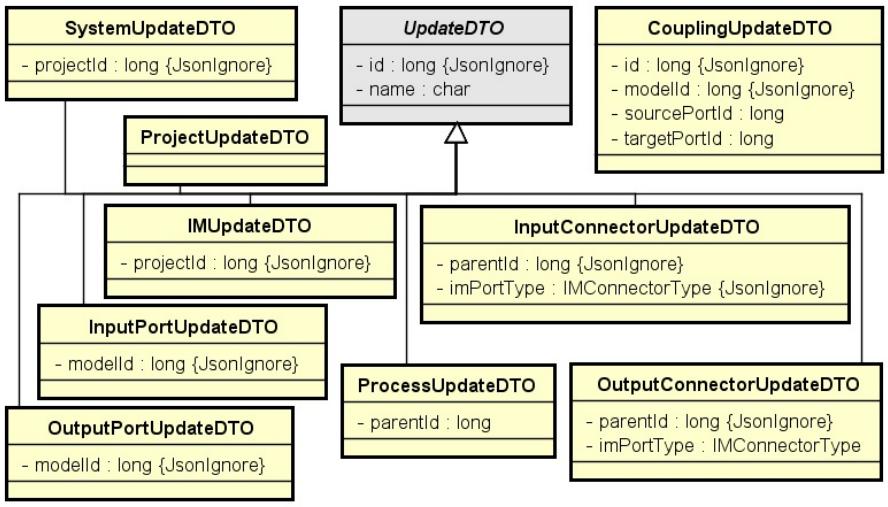
Table 5.1: URL Signatures for Different REST APIs of the DEVS-IM Framework.

Element	URL Signatures
Project	/IM/Projects[/{projectId}]
IM	/IM/Projects/{projectId}/IMS[/{imId}[/GenerateCode]]
Model	/IM/Models/{modelId}
Input Connector	/IM/Models/{modelId}/InputConnectors[/{connectorId}]
Output Connector	/IM/Models/{modelId}/OutputConnectors[/{connectorId} [/Functions[/functionId]]]
Process	/IM/Models/{modelId}/Processes[/{processId}]
Task	/IM/Models/{modelId}/Tasks[/{taskId}]
Input Port	/IM/Models/{modelId}/InputPorts[/{portId}]
Output Port	/IM/Models/{modelId}/OutputPorts[/{portId}]
Coupling	/IM/Models/{modelId}/Couplings[/{couplingId}]
System	/IM/Projects/{projectId}/Systems[/{systemId}]
Component	/IM/Systems/{systemsId}/Components[/{componentId}]
Function	/IM/Components/{componentId}/Functions[/{functionId}]

A portion of the class diagram for the "Service" and the "Data Access" layers of the DEVS-IM is shown in Figure 5.14. Each service class in the `Service` package has some association relation with other service classes, and one association relation with the corresponding repository in the `Data Access` package (association from `ModelService` and `IMService` classes to the `ModelRepository` and `IMRepository` classes are shown in the diagram). The interfaces define the required methods which the corresponding classes must implement. Other details of the classes in the `Services` and `Data Access` packages are omitted for brevity.



(a)



(b)

Figure 5.13: A Portion of the Class Diagram for the DTOs in the DEVS-IM. (a) To Retrieve Data. (b) To Insert or Update Data.

A sequence diagram scenario for a client inserting an *IM* element is shown in Figure 5.15. The incoming message 1 by the `ui` object is processed by the `imc` object. The *id* of the project sets in step 2. In step 3, a message is invoked on the `ims` object to insert a new interaction model. The `ims` object checks some validation in step 4 (i.e., the *name* attribute cannot be null or empty, having a valid *id* for the project, and prevent duplicate names for the interaction model). Then, the `DTO` object maps

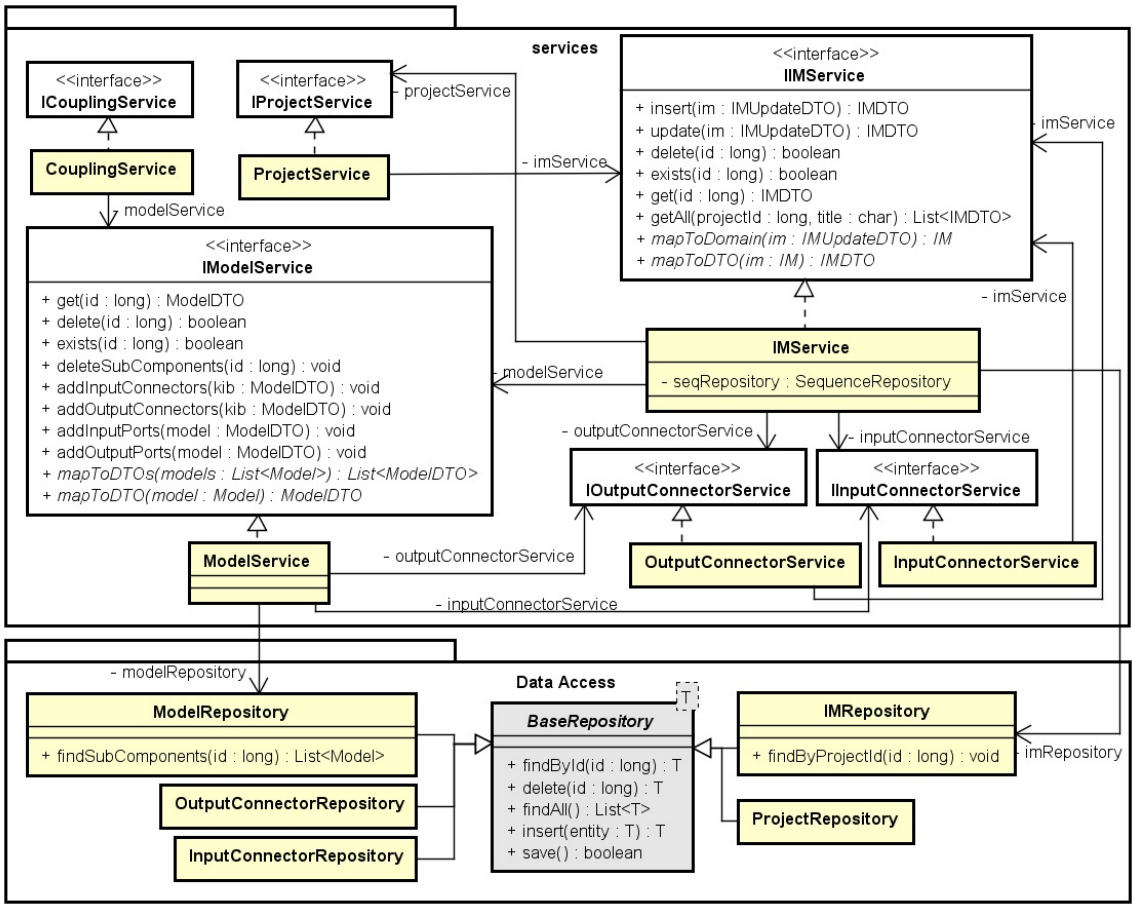


Figure 5.14: A Partial Class Diagram for the Service and Data Access Layers of the DEVS-IM Framework.

to a `Domain` object, and a new valid sequential unique *id* is set. The `ims` object invokes message 7 on the `imr` object for inserting the interaction model. The `imr` object invokes message 8 on the `db` object (i.e., MongoDB database) and returns the inserted data to the database. Finally, the `im` object (the `Domain` object) maps to the `IMDTO` object and returns as a result in step 9. The result returns to the `imc` and `ui` objects, consequently.

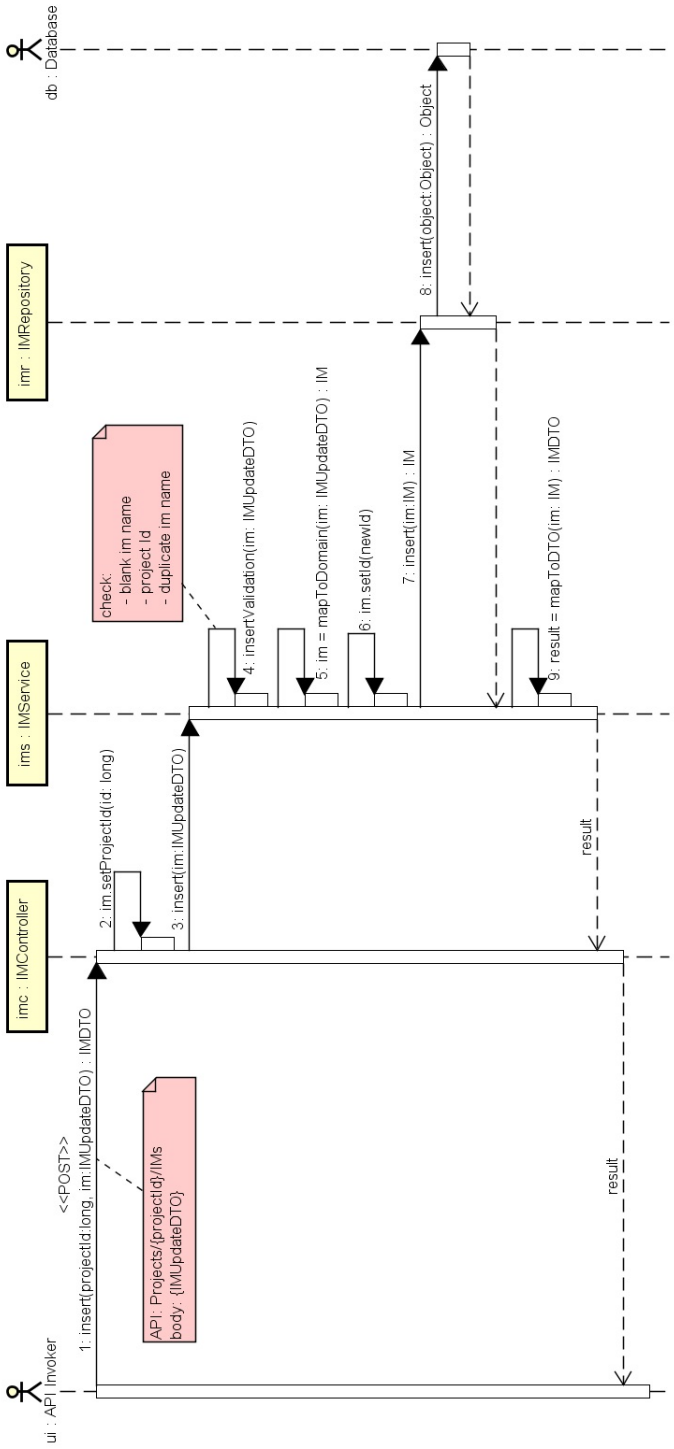


Figure 5.15: A Sequence Diagram to Insert an *IM* Element via the DEVS-IM Framework.

5.3.3 Model Verification

The applied verification on each received request guarantees that the stored model in the database is always a valid model from the structure point of view. Different verifications are defined for CRUD operations on each element. A proper error message is returned in JSON format if the request breaks any verification rule of defining the structure of the model. Otherwise, a JSON object with proper attributes to show a successful operation will return as the response. For example, the framework prevents duplicate names for a type of element (e.g., *Process*, *Task*, or *Input/Output Port*) in the same scope (the same hierarchy level).

Figure 5.16 illustrates a verification flowchart for inserting a Coupling element in the DEVS-IM framework. Suppose an API caller requests the URL "/IM/Models/1/Couplings" via a POST method with "`{\sourcePortId": 1, \targetPortId": 2}`" as the body of the request. After routing to a proper service (see Figure 5.12), the verification starts by checking the modelId. It retrieves a model with id that equals one (based on the received id in the URL). The first check will be passed if the model exists. Otherwise, an exception with the message "invalid modelId" returns to the caller. In the second step, the type of the model must be checked. The *IM* and *Process* elements (coupled DEVS models) can have coupling. So, an exception with the message "invalid modelType" returns to the caller if the type is not valid. In the third step, the model is checked for trying to insert repetitive coupling. So, an exception with the message "repetitive coupling" returns if the coupling already exists in the current model. The next step is finding the source and target models of the coupling (using the `sourcePortId` and `targetPortId` in the body of the request) to verify the coupling based on the DEVS specification. A coupling is valid if it is an *External Input Coupling* (EIC): coupling from the input port of the current

model to the input port of a sub-model, *External Output Coupling* (EOC): coupling from the output port of a sub-model to the output port of the current model, or *Internal Coupling* (IC): coupling from the output port of a sub-model to the input port of another sub-model (see Section 2.2). An exception with the message "invalid coupling" returns if none of the conditions are satisfied (specified by numbers 4 to 6 in Figure 5.16). The model in the database will be updated (add the coupling to the model) if all checking are passed.

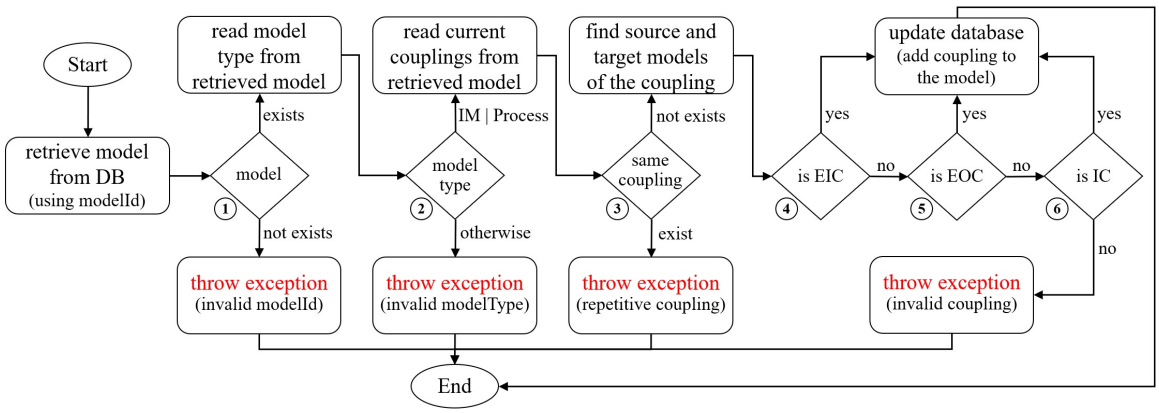


Figure 5.16: Verification Flowchart for Inserting a *Coupling* Element in the DEVS-IM Framework.

Furthermore, a tree structure must be defined for the "Interaction Model" and the "External System Schema" sections in Figure 5.6. So, the framework always checks not to have a loop in defining a model. In other words, a model cannot be a self-child. Otherwise, it makes an infinite loop in the defined model. Also, the *name* attribute of all elements (except the *Port* element) must follow the pattern "`[a-zA-Z][a-zA-Z_$0-9]`" because the name of these elements will be used in the code generation phase to define the name of the generated packages and files (the code generation is explained in the next section).

5.3.4 Database Specification

Figure 5.17 illustrates the database schema in the DEVS-IM design. There are four collections with *id* as the primary key. The `pk`, `fk`, and `dk` show the primary key, foreign key, and destination key in a collection/relation, respectively. Mandatory values are indicated by the star. The `projects` collection stores *Project* elements (see Figure 5.11). All atomic and coupled models (i.e., all hierarchy of *IM*, *Logic*, *Task*, *Process*, and *Connector* elements) defined in an interaction model are stored in the `models` collection. The ports and couplings of a model are stored as nested collections. The couplings collection for an atomic model would be empty. The data of the external system interfaces (see Figure 5.11) are stored in the `systems` and `components` collections. The ports and couplings in the `models` collection and the functions in the `components` collection are defined using one-to-many relationships with embedded documents. The hierarchy structure for the models and components is defined using a one-to-one relationship with document references (using *parentId* field). The rest relationships are defined using one-to-many relationships with document references.

The stored data in the database (specifically the `models` collection) will be used to generate the skeleton of a complete project in the DEVS-Suite simulator to define the behavior of the atomic models in the Java programming language. The predefined behavior of the *Logic* and *Connector* elements define during the code generation. Thus, the modeler can define the *Task* and *Function* elements' behavior. The test, debug, and visualization features of the DEVS-Suite simulator can also be used to validate the interaction model's correctness.

As mentioned before, the DEVS-Suite simulator is used in this research to simulate the interaction model. The structure of a models is defined using the DEVS-IM REST APIs and the data are stored in the MongoDB database. The required classes (in

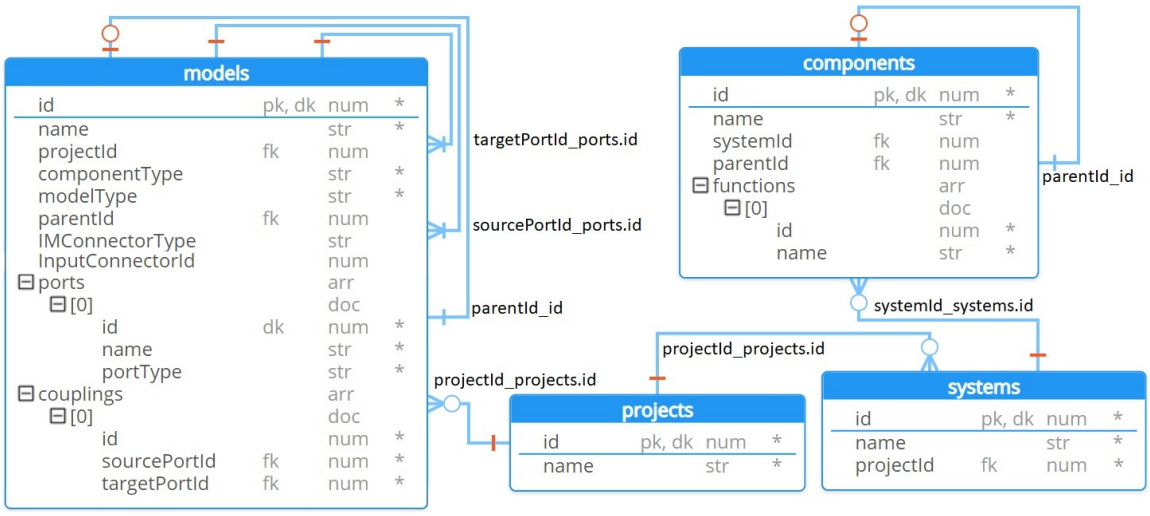


Figure 5.17: Database Schema to Store the DEVS-IM Models.

Java programming language) for the DEVS-Suite simulator must be generated to add the behavior for the atomic models (Task and Function elements) by the modeler.

WEN Simple Example

The DEVS-IM model for the exemplar WEN system (see Figure 4.1 and Figure 4.2) is defined using 88 APIs with the POST method from Table 5.1 (one *Project*, two *System*, 10 *Component*, eight *Function*, one *IM*, seven *Input Connector*, eight *Call Output Connector*, two *Process*, five *Task*, 22 *Port*, and 22 *Coupling* elements). Figure 5.18 presents a schematic view (drawn manually) of the defined DEVS-IM model. Two interfaces are defined for the water and energy external systems, and they are presented in a tree structure. The water interface has a *run()* function to execute the WEAP model and two sub-components for the demands and supplies. The required functions to get/set the values from/to the external systems are defined under the correspondence components (i.e., the *setFlow()* function under the "PowerPlant" component and the *getFlow()* function under "Canal1" and "Canal2" components). The same approach is applied to the energy interface to define its demands and supplies. The

tree structure of an interface is not required to be the same as the structure of the external correspondence system. Different interfaces at different levels of abstraction can be defined for an external system (based on the modeler's approach). Compare the defined interfaces in Figure 5.18 with the defined model in the WEAP and LEAP systems in Figure 4.2. For example, the "Canal1" element in Figure 5.18 (under the supplies of the water interface) is a reference to the *Transmission Link* entity from the "River" to the "PowerPlant" entities in the WEAP model (see Figure 4.2). It is simplified to one element in the interface for the external system in Figure 5.18 (instead of defining the whole hierarchy presented in the WEAP model) and hides some details which are not important or not used in the interaction model.

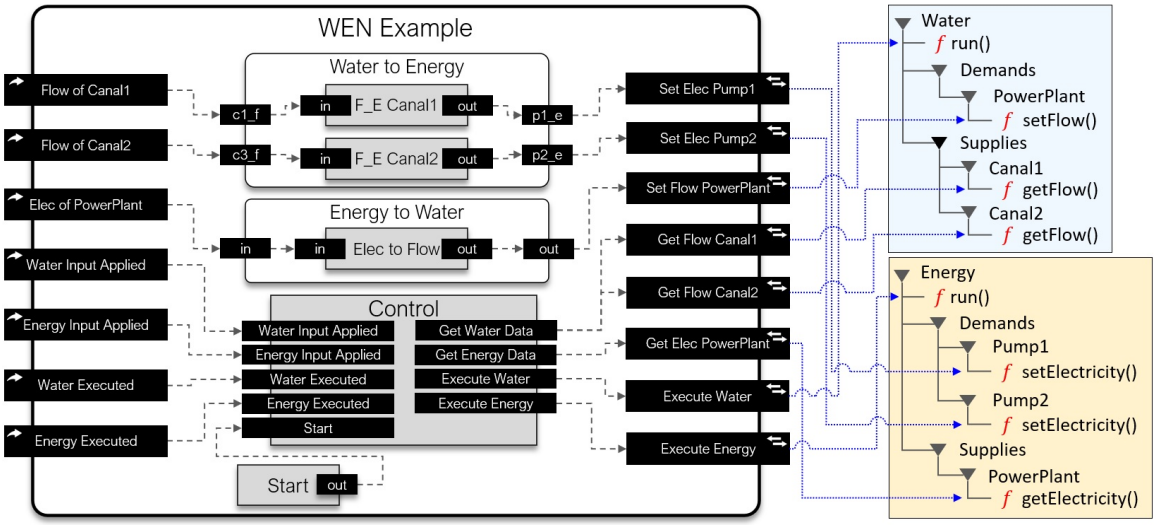


Figure 5.18: A Schematic Diagram of the Defined DEVS-IM Model for the Exemplar WEN Model.

The interaction model is defined using seven *Input Connector*, eight *Call Output Connector*, two *Process*, and five *Task* elements (the left side in Figure 5.18). Each output connector connects to a specific function of the external system schemas. There are couplings between the output connectors and input connectors, but they are not presented in Figure 5.18 for simplification and focus on the connection between the IM and external system schemas. Indeed, all output connectors are coupled with a

specific input connector (e.g., the "Get Flow Canal1" output connector is coupled with the "Flow of Canal1" input connector) to send the results. The "Control" element controls the synchronization between different parts of the interaction model and the external systems. It also manages the concurrency/execution of the three main systems (i.e., water, energy, and interaction model). The Process elements contain the *Task* elements to transform the *Flow*/*Electricity* values from the water/energy model to the *Electricity*/*Flow* values for the energy/water model (i.e., implementing the Formulas 4.1 and 4.4). The execution mechanism is explained in the next sections.

5.3.5 DEVS-Suite Simulator Code Generation

The code generator of the DEVS-IM framework supports translating the models that are stored in the database to source code for target simulation and markup languages. Figure 5.19 presents the code generation schema from the DEVS-IM model to generate meaningful java classes, as the skeleton of a complete project, in the DEVS-Suite simulator. The stored model in the database (defined via the REST APIs) is fetched and passed to the "Template Files" (using the "Template Models") to generate multiple packages and files. A set of "stg" files (String Template Group) are used as the template to define the skeleton of the final generated classes for different types of models (Parr, 2022). The "Template Models" contains a set of `Data Transfer Object` classes to transfer data from the database to the "Template Files". A package with the same name (as the project name) will be generated and contains all files related to the project. For each project, the code generator generates one package for each IM model and one package for each system (*System_1 Adaptor* to *System_n Adaptor* in Figure 5.19). As shown in Figure 5.19, the generated package for the IM model contains the classes for the *IM*, *Process*, *Task*, and *Connector* elements

(see Figure 5.8). Also, the generated package for each external system contains the classes for *System*, *Component*, and *Function* elements.

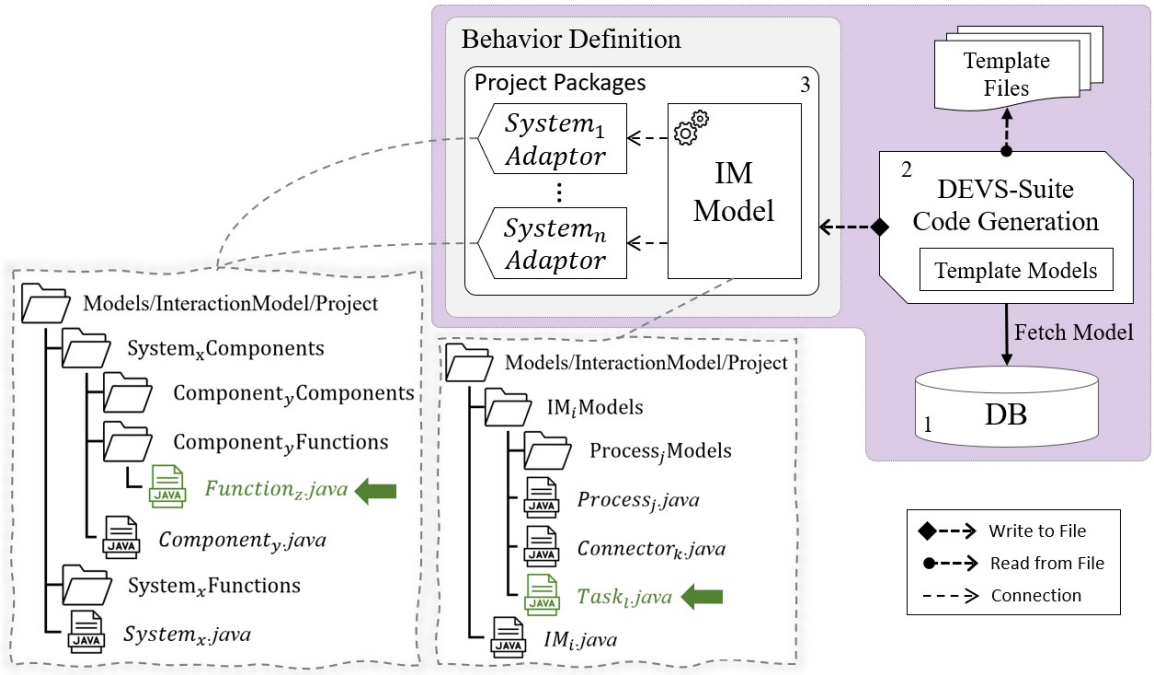


Figure 5.19: The Code Generation Schema from the DEVS-IM Model to the DEVS-Suite Simulator.

From the DEVS specification point of view, the *IM* and *Process* elements are coupled models (but the *IM* does not have any input or output ports), and the *Connector* and *Task* elements are atomic models. The coupled models are presented using a java file to define the structure of the model and a package containing the sub-models. For example, in Figure 5.19, the "$IM_i.java$" file and the "$IM_iModels$" package define the i^{th} *IM* element. Also, the "$Process_j.java$" file and the "$Process_iModels$" package define the j^{th} *Process* element. The atomic models are presented using a java file to define the structure and the behavior of the model. For example, the "$Connector_k.java$" and "$Task_l.java$" in Figure 5.19 define the k^{th} Connector and the l^{th} Task elements.

To define a tree structure as an interface for the external system, the *System* and *Component* elements are presented using a java file to define the system/component and two packages for its sub-components and functions. For example, in Figure 5.19, the "$Component_y.java$" file with the "$Component_yComponents$" and "$Component_yFunctions$" packages define the y^{th} Component element (the same files and packages structure is used for the System element). The *Function* elements are presented using a java file (to define the behavior). For example, the "$Function_z.java$" in Figure 5.19 defines the z^{th} *Function* element. The code generator defines the entire structure of the interaction model. The behavior of the connectors is generated, and the user needs to complete the behavior for *Task* and *Function* elements (the green files pointed by green arrows in Figure 5.19).

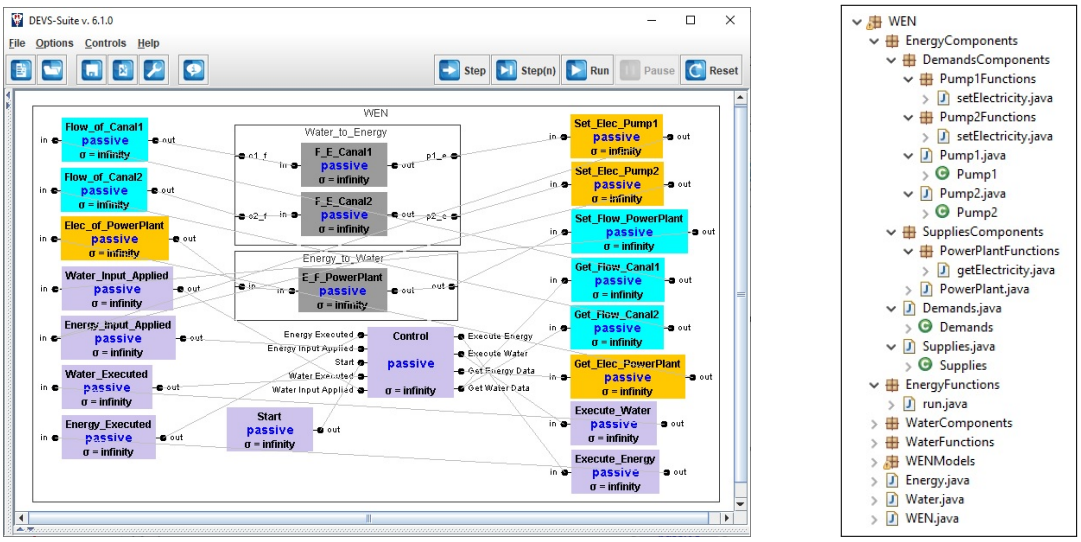
The "Interaction Model" and "External System Schema" sections of Figure 5 have been explained. Now, the communication between the "Function" part and the "External Systems" section in Figure 5.6 needs to be defined. There exist a wide variety of tools for modeling and simulating the external systems (i.e., the water and energy systems). As a generic approach, legacy and object-oriented software systems can be encapsulated as services in Service Oriented Architecture (SOA) paradigm. Various approaches have been proposed to transform legacy software systems into integrable with other software systems (Bisbal et al., 1999). One of these approaches is "wrapping", where any proprietary legacy software system with input/output API (e.g., WEAP and LEAP systems) can be encapsulated inside other software systems (Sneed, 2006; Sneed et al., 2006). Indeed, individual functions in the legacy software are wrapped into web services. This research follows the rationale and the general approach of transforming a legacy system into flexible service-oriented software frameworks in addition to component-based modeling and simulation.

The WEAP/LEAP system is appealing to domain experts from the standpoint of ease of use for rapid model development. In our previous publication (Fard & Sarjoughian, 2021a), the WEAP entities, input and output variables, and their data are represented using the Ecore meta-modeling approach, where each proxy model component corresponds to a WEAP entity. The Ecore presents a well-defined componentized specification for the WEAP legacy modeling and simulation system. These components are used in a flexible service-oriented framework. The outcome is the Componentized WEAP RESTful framework. The framework helps to consider a set of component models instead of thinking about a group of shared variables (belonging to different entities) that are used in mass-balanced equations. Also, the REST APIs ease the use of the WEAP system in modern computing platforms, including its integration with other tools to model and simulate more complex systems, like the WEN system. Every model entity developed in the WEAP system is automatically extracted and included as a componentized model in the Componentized WEAP RESTful framework. In the following section, the Componentized WEAP RESTful framework is presented from the communication perspective. The same approach is applied to the LEAP system.

WEN Simple Example

After using the Code Generation module of the DEVS-IM framework for the defined model in the previous step, the SimView window of the DEVS-Suite simulator and the generated packages (the focus of the Energy External System interface) for the exemplar WEN model are presented in Figure 5.20 (compare it with Figure 5.18 and Figure 5.19). The *Process* elements are converted to the coupled DEVS models. The *Task* and *Connector* (input and output) elements are converted to the atomic DEVS models. The purple atomic models are used to control the execution (concurrency

and synchronization), and the blue and yellow atomic models are connected to the water and energy models, respectively. Each connector has one input port (named "in") and one output port (named "out"), which are added automatically during the model definition phase. The couplings between the output and input connectors are presented in this view (i.e., the coupling from the "out" output port of an output connector to the "in" input port of an input connector).



(a) (b)

Figure 5.20: Automatic Generated Code for the Exemplar WEN Model. (a) The DEVS-Suite SimView. (b) Generated Packages, Focus on the Energy External System Interface.

Considering the defined water and energy models in the WEAP and LEAP tools in Figure 4.2, presents the JSON results of calling some APIs to get data for the defined entities, variables, and simulation values. Given the execution of the WEAP/LEAP tool and the Componentized WEAP/LEAP framework, Figure 5.21a shows the result of calling "/Energy/WENExample/Demands" via the GET HTTP method. The result contains a list of demands defined in the "WENExample" project (see Figure 4.2). Each demand has *name*, *id*, *parentId*, and *order* properties. Figure 5.21b shows the result of calling "/Water/WENExample/Demands/PowerPlant/Outputs" via the GET

HTTP method. The result contains a list of all output variables of the "PowerPlant" demand site in the "WENExample" project. Each variable has *name*, *unit*, *min*, *max*, *isReadOnly*, and *isUserDefined* properties. Figure 5.21c presents the retrieved dummy data from the water model for the "Water Demand" variable of the "PowerPlant" demand site for "CurrentAccount" scenario and data are filtered for 2010 values (by calling URL= "/Water/WENExample/DemandSites/PowerPlant/Outputs/Water Demand/Current Account?startYear=2010,endYear=2010"). The data section has four values for different timesteps (i.e., seasons). The similar API from the LEAP system, Figure 5.21d, returns one value per year because it is assumed to have yearly time granularity for the energy system in the WEN example (see Figure 4.1).

5.3.6 Behavior Definition

Calling the Componentized WEAP REST APIs must be defined in the Function elements of the interaction model (see Figure 5.6). A base package named "`core`" is implemented in the DEVS-Suite simulator, which contains the super abstract classes for all element types of the DEVS-IM framework (see Figure 2 and Figure 3 in (Fard & Sarjoughian, 2021b)). A partial behavior for the `Task` abstract class is defined in the `core` package. The inherited class from the `Task` class must complete the behavior. The `Input/Output Connector` abstract classes in the `core` package are inherited from the `Task` class, and their default behavior is defined, as well. The design of the classes in the core package has the flexibility to define a new type of element by the modeler (e.g., a new connector element).

The *Task* element in the DEVS-IM design serves the purpose of the *Transformation* element in the Algorithmic-IM approach. The main difference between these two elements is the capability to set the time advance value in the *Task* element. Also,

Figure 5.21: (a) JSON Result of Calling URL = "/Energy/WENExample/Demands". (b) JSON Result of Calling URL= "/Water/WENExample/DemandSites/PowerPlant/Outputs". (c) JSON Result of Calling URL = "/Water/WENExample/DemandSites/PowerPlant/Outputs/Water Demand/ CurrentAccount?startYear=2010,endYear=2010". (d) JSON Result of Calling URL = "/Energy/WENExample/Demands/Pump1/Inputs/Energy Intensity/CurrentAccount?startYear=2010,endYear=2010".

a modeler can define other new elements (derived from the `IMAtomicModel` class in Figure 5.9) which have different behavior compared to the *Task* element. The DEVS specification of the *Task* element is defined in Listing 5.1. The modeler must define the input and output port names and values. The state variable is defined using the `phase`, `sigma`, `queue`, `current message`, and `active time` attributes. The "Active" and "Passive" are two valid values for the `phase` attribute. The `queue` attributes to store the input messages received on the input ports if the model is processing an-

other input. The `current message` attribute indicates the message that the model is processing. The `active time` attribute indicates the required time to process input (amount of time for being in the "Active" phase). The default value for the `active time` is *zero*. However, it is an input parameter of the `Task` class in the DEVS-Suite simulator, so it can be set to different values for different task instances. The external transition function sets the active time and input message to the sigma and current message state variables if a model is in the "Passive" phase. Otherwise, the input message is added to the queue variable, and the sigma is updated. The internal transition function starts processing the next input message if the queue is not empty. Otherwise (the queue is empty), the phase is changed to "Passive". The actual data transformation must be defined in the *perform(cv,Y)* method. The output function is responsible for calling the *perform* method and passing the required parameters to it (the current message and the output set). The specifications for the remaining functions are straightforward. By having these specifications for the *Task* element, a modeler who is not an expert in the DEVS formalism can define the behavior of the transformations (in the *perform* method) without going deep into different functions of the atomic DEVS model. Some useful predefined elements (such as *Queue*, *Stack*, *Random Generator*, *Periodic Generator*, and so on) can be inherited from the `Task` class to have simpler and faster modeling.

Listing 5.1: Parallel DEVS Specification of the *Task* Element.

$$M_{Task} =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$$
$$IPorts = \{\dots\}, OPorts = \{\dots\}$$
$$X = \{(ip, v) | ip \in IPorts, v \subset Values\}$$
$$Y = \{(op, v) | op \in OPorts, v \subset Values\}$$
$$S = \{\text{“}Passive\text{”}, \text{“}Active\text{”}\} \times \mathbb{R}_0^{+\infty} \times Queue < (ip, v) > \times (ip, v) \times \mathbb{R}_0^{+\infty}$$
$$s_0 = \begin{cases} (\text{“}Passive\text{”}, Infinity, \varnothing, \varnothing, 0) & initPhase == \text{“}Passive\text{”} \\ (\text{“}Active\text{”}, ActiveTime(), \varnothing, \varnothing, 0) & otherwise \end{cases}$$
$$\delta_{ext}((phase, \sigma, q, cv, at), e, (p, v)) =$$

$$\begin{cases} (\text{“}Active\text{”}, at, \varnothing, (p, v), at) & initPhase == \text{“}Passive\text{”} \\ (\text{“}Active\text{”}, \sigma - e, q.add((p, v)), cv, at) & otherwise \end{cases}$$

$$\delta_{int}(phase, \sigma, q, cv, at) = \begin{cases} (\text{“}Active\text{”}, at, q, q.pop(), at) & q.size() > 0 \\ (\text{“}Passive\text{”}, Infinity, \varnothing, \varnothing, at) & otherwise \end{cases}$$

$\delta_{con}((phase, \sigma, q, cv, at), e, (p, v)) = \delta_{ext}(\delta_{int}(phase, \sigma, q, cv, at), 0, (p, v))$
$\lambda(\text{“}Active\text{”}, \sigma, q, cv, at) = perform(cv, Y)$
$ta(phase, \sigma, q, cv, at) = \mathbb{R}_0^{+\infty}$

In addition, the *Logic* element is an atomic model with a specific behavior that does not increase the simulation clock. It can be used to define some logical operations. For example, the *Junction* element sends data on the input/s to all output ports, the *Choice* element sends the input data to one of the outputs based on some condition, and the *Synchronization* element to sync the inputs and send them on output/s. As an example, Listing 5.2 presents the specification for the *Choice* element. The incoming message will send to one of the output ports (in the *perform()* method) based on a set of conditions. No explicit concept of the logic component was used in the Algorithmic-IM. Like the *Task* element, the *Logic* elements can be defined by the user.

Listing 5.2: Parallel DEVS Specification of the *Choice* Element.

$M_{Task} =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$
$IPorts = \{\dots\}, OPorts = \{\dots\}$
$X = \{(ip, v) | ip \in IPorts, v \subset Values\}$
$Y = \{(op, v) | op \in OPorts, v \subset Values\}$
$S = \{\text{“}Passive\text{”}, \text{“}Active\text{”}\} \times \mathbb{R}_0^{+\infty} \times Set < condition, p >$
$s_0 = (\text{“}Passive\text{”}, Infinity, \varnothing)$
$\delta_{ext}((phase, \sigma, s), e, (p, v)) = (\text{“}Active\text{”}, 0, s)$
$\delta_{int}(phase, \sigma, s) = (\text{“}Passive\text{”}, Infinity, s)$
$\delta_{con}((phase, \sigma, s), e, (p, v)) = \delta_{ext}(\delta_{int}(phase, \sigma, s), 0, s)$
$\lambda(\text{“}Active\text{”}, \sigma, s) = perform(s, Y)$
$ta(phase, \sigma, s) = \mathbb{R}_0^{+\infty}$

One input connector type is defined in the DEVS-IM framework (i.e., the *Transient Input Connector* element) (Fard & Sarjoughian, 2021b). Its DEVS formal spec-

ification is presented in Listing 5.3. All predefined input/output connectors in the DEVS-IM framework have one input port (named "in") and one output port (named "out"). The behavior is defined as transiently transferring any input message on the "in" input port to the "out" output port of the model (`active time` is *zero*). The state variable is defined using the `phase`, `sigma`, and the received `Values` on the message.

Listing 5.3: Parallel DEVS Specification of the *Transient Input Connector* Element.

$M_{TransientOutput} =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$
$IPorts = \{in\}, OPorts = \{out\}$
$X = \{(ip, v) | ip \in IPorts, v \subset Values\}$
$Y = \{(op, v) | op \in OPorts, v \subset Values\}$
$S = \{\text{“}Passive\text{”}, \text{“}Active\text{”}\} \times \mathbb{R}_0^{+\infty} \times Values$
$s_0 = (\text{“}Passive\text{”}, Infinity, \varnothing)$
$\delta_{ext}((phase, \sigma, m), e, (p, v)) = (\text{“}Active\text{”}, 0, v)$
$\delta_{int}(phase, \sigma, m) = (\text{“}Passive\text{”}, Infinity, \varnothing)$
$\delta_{con}((phase, \sigma, m), e, (p, v)) = \delta_{ext}(\delta_{int}(phase, \sigma, m), 0, (p, v))$
$\lambda(\text{“}Active\text{”}, \sigma, q, cv, at) = (\text{“}out\text{”}, m)$
$ta(phase, \sigma, q, cv, at) = \mathbb{R}_0^{+\infty}$

Three types of output connector elements are defined in the DEVS-IM framework; *CallOutput*, *TransientOutput*, and *QueueOutput* connectors. The *TransientOutput* connector immediately sends data to the external systems. The received data on the input can be sent to a list of *Function* elements (using the *exec(. . .)* method of the selected function). The DEVS specification for *TransientOutput* connector is presented in Listing 6. The output function corresponds to sending the incoming message to all destinations. For example, the data for the input variables of the WEAP or LEAP models can be set using the *TransientOutput* connector (via Componentized WEAP and Componentized LEAP APIs).

Listing 5.4: Parallel DEVS Specification of the *Transient Output Connector* Element.

$M_{Task} =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$
$IPorts = \{in\}, OPorts = \{out\}$
$X = \{(ip, v) | ip \in IPorts, v \subset Values\}$
$Y = \varnothing$

$S = \{\text{“}Passive\text{”}, \text{“}Active\text{”}\} \times \mathbb{R}_0^{+\infty} \times Values$
$s_0 = \text{“}Passive\text{”}, Infinity, \varnothing$
$\delta_{ext}((phase, \sigma, m), e, (p, v)) = (\text{“}Active\text{”}, 0, v)$
$\delta_{int}(phase, \sigma, m) = (\text{“}Passive\text{”}, Infinity, \varnothing)$
$\delta_{con}((phase, \sigma, m), e, (p, v)) = \delta_{ext}(\delta_{int}(phase, \sigma, m), 0, (p, v))$
$\lambda(\text{“}Active\text{”}, \sigma, m) = destinations(d).exec(m)[\forall d \in \mathbb{N}_0^+, d < destinations.length]$
$ta(phase, \sigma, q, cv, at) = \mathbb{R}_0^{+\infty}$

The *CallOutput* connector is connected to one *Function* element. The defined behavior for this model is sending the received message on the “in” input port to the connected *Function* element. Then, execute the *exec(. . .)* method of the *Function* element and put the result of the execution to the “out” output port of the connector (which will be sent to an input connector). The DEVS formal specification of the *CallOutput* connector is presented in Listing 5.5. Like the *TransientInput* connector, the state variable is defined using the `phase`, `sigma`, and the received `Values` on the message. The destination in the output function refers to the connected *Function* element to the model. The *TransientOutput* connector is connected to a set of *Fucntion* elements. The defined behavior for this model is sending the message received on the “in” input port to all connected *Function* elements and executing their *exec(. . .)* method. The *QueueOutput* connector queues the message received on the “in” input port to be read in the future.

Listing 5.5: Parallel DEVS Specification of the *Call Output Connector* Element.

$M_{CallOutput} =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$
$IPorts = \{\ldots\}, OPorts = \{\ldots\}$
$X = \{(ip, v) | ip \in IPorts, v \subset Values\}$
$Y = \{(op, v) | op \in OPorts, v \subset Values\}$
$S = \{\text{“}Passive\text{”}, \text{“}Active\text{”}\} \times R_0^{+\infty} \times Values$
$s_0 = \text{“}Passive\text{”}, Infinity, \varnothing$
$\delta_{ext}((phase, \sigma, m), e, (p, v)) = (\text{“}Active\text{”}, 0, v)$
$\delta_{int}(phase, \sigma, m) = (\text{“}Passive\text{”}, Infinity, \varnothing)$
$\delta_{con}((phase, \sigma, m), e, (p, v)) = \delta_{ext}(\delta_{int}(phase, \sigma, m), 0, (p, v))$
$\lambda(\text{“}Active\text{”}, \phi, m) = (\text{“}out\text{”}, destination.exec(m))$
$ta(phase, \phi, m) = R_0^{+\infty}$

Figure 5.22 illustrates an imaginary example in the DEVS-IM (the visualization is drawn manually). The created model by the modeler is presented at the top layer, and the equivalent DEVS model is shown at the bottom. The input and output connectors must be connected to the system interfaces (which are not presented in the diagram). From the modeler perspective, there are three *InputConnector*, three *OutputConnector*, one *Process*, one *Task*, one *Junction* elements, and the coupling between them. The connectors named "out1", "out2", and "out3" of the "IM" interaction model are *TransientOutput*, *QueueOutput*, and *CallOutput* elements, respectively. Suppose the input connector "in3" of the "IM" component is selected in the output connector "out3" for the input attribute (see Figure 5.9). In the equivalent DEVS model, all components are presented as atomic or coupled models. There are eight atomic models and one coupled model in the "IM" component (the "Process 1" coupled model has sub-elements, as well).

As described before, all elements in the DEVS-IM design are derived from the atomic or coupled DEVS models. Thus, the interaction model is a parallel DEVS model (with some specific atomic models as the connectors to communicate with the outside world). Consequently, the DEVS simulation protocol is used to simulate the interaction model. To simulate the DEVS models, a hierarchy of simulator objects that mirrors the hierarchical tree structure of the DEVS model is used. There is a DEVS simulator corresponding to each atomic model and a DEVS coordinator corresponding to each coupled model (see Section 2.2). A root coordinator oversees controlling the executions of all atomic and coupled simulators. The simulators and coordinators are responsible for the correct simulation of coupled models. A key advantage of using a well-defined simulation protocol is that it allows a simulator to execute models independent of their specific behaviors.

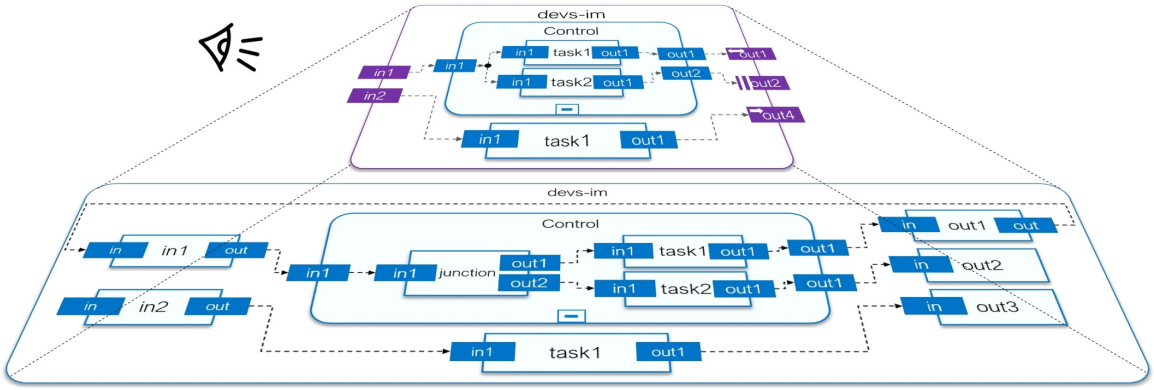


Figure 5.22: An Example DEVS-IM Viewed as a DEVS Model.

5.3.7 Execution Control

Different modeling formalisms in a heterogeneous modeling environment provide their own approaches for modeling behavioral specification and execution protocol. Consequently, the simulation time management and synchronization mechanisms of different modeling formalisms can be so distinct that the dynamic interaction between the disparate models may result in unexpected errors without a well-defined composite behavioral specification or a well-designed coordination execution protocol. The DEVS-IM framework is grounded in system theory and component-based modeling. The model specification and execution protocol are separated in the DEVS-IM framework (based on using DEVS formalism). It supports the model reusability, flexibility, and maintainability traits essential for developing realistic simulations (such as coupled energy and water systems). The Parallel DEVS supports concurrent execution of the DEVS models. Therefore, the DEVS-IM model may send messages to disparate external systems simultaneously. Also, the DEVS-IM model must be capable of dealing with concurrent messages from external systems.

The DEVS-IM simulator framework supports synchronous communications/executions. Based on the defined execution control in the interaction model, multiple requests from the external system/s (e.g., send data, receive data, run the external

simulation, and so on) can be invoked sequentially or in parallel. The time inside the interaction model would not be elapsed while it is waiting for the result/s from the external system/s. In sequential execution control, the interaction model requests a specific action from an external system and waits for its result. Then, another request from the same or another external system can be invoked. In the parallel execution control, the interaction model requests from multiple external systems simultaneously and waits for all results. A hybrid execution can be defined in the interaction model, meaning that some intervals follow the sequential execution, and others follow the parallel execution (based on the defined execution control for the model). Even though asynchronous execution can be supported by the programming language of the DEVS-Suite simulator (the Java programming language), the simulator cannot guarantee a robust execution for the asynchronous behavior.

WEN Simple Example

Figure 5.23 shows a pseudo-code to control the execution for the exemplar WEN model. It has five input ports and four output ports. Lines 1-5 define the local variables with their initial values. Lines 6-8 must be implemented by all Task elements to define the behavior of the model. In this example, two sub-procedures are called to read the messages on the input ports and produce some messages on the output ports. The *CheckInputMessages* procedure (lines 9-20 in Figure 5.23a) sets the local variables based on the read messages on the input ports. For example, the control starts executing by receiving a message on "Start" input port; Or it increases the *EnergyInputApplied* variable for one unit by receiving a message on the port "Energy Input Applied"; And so on.

The *FillOutputMessages* procedure can have behavior like Figure 5.23b to have a sequential execution for the external systems. In the sequential execution, the water

and energy models run alternatively (see toggling the *isWaterExecuted* and *isEnergyExecuted* local variables). The *WaterInputApplied* and *EnergyInputApplied* variables check that all output values are applied to the external systems (i.e., `set_Elec_Pump1` and `set_Elec_Pump2` for the energy system, and `set_Flow_PowerPlant` for the water system). Figure 5.23c presents the parallel execution of the water and energy systems. These are two samples of the sequential and parallel executions applied by the Control model. Any other execution can be implemented by the user (based on the input and output messages and the coupling from Control to the other models). Obviously, different execution controls for the same systems can produce different results.

Figure 5.24 illustrates the sequence diagram of executing the water system (WEN project of the WEAP tool) in the defined DEVS-IM model. Compare the models (and the color of the objects) in the sequence diagram with the models in Figure 5.20. In step 1, a message is sent from `cont` object (an instance of the `Control` atomic model) to the `ew` object (an instance of the `Execute_Water` output connector) via their coupling. In step 2, the `ew` object calls the *exec()* method of the `runWEAP` object (an instance of the `run` function). In step 3, an API from the Componentized WEAP framework is called to run the "WEN" project. In step 4, the Componentized WEAP framework executes the project and returns the *True* value in JSON format. Then, the `runWEAP` object puts the received JSON result in a message of type Entity and returns the message to the `ew` object. All communications from step 2 up to this point were a synchronous process. In step 5, the `ew` object sends the received message to the `we` object (an instance of the `Water_Executed` input connector) via their coupling. Consequently, the received message is sent to the `cont` object in step 6. Finally, the *isWaterExecuted* variable of the `cont` object is set to *True* (see Figure 14 5.23a). The same steps apply to read/write values from/to a specific variable of an entity in the WEAP and/or LEAP models.

```
1: started ← False
2: isWaterExecuted ← False
3: isEnergyExecuted ← False
4: WaterInputApplied ← 0
5: EnergyInputApplied ← 0

6: procedure PERFORM(inputMsg, outputMsg)
7:     CHECKINPUTMESSAGES(inputMsg)
8:     FILLOUTPUTMESSAGES(outputMsg)

9: procedure CHECKINPUTMESSAGES(inputMsg)
10:    for each (msg in inputMsg) do
11:        if (msg.isOnPort("Start")) then
12:            started ← True
13:        else if (msg.isOnPort("Water Executed")) then
14:            isWaterExecuted ← True
15:        else if (msg.isOnPort("Energy Executed")) then
16:            isEnergyExecuted ← True
17:        else if (msg.isOnPort("Water Input Applied")) then
18:            WaterInputApplied ← WaterInputApplied + 1
19:        else if (msg.isOnPort("Energy Input Applied")) then
20:            EnergyInputApplied ← EnergyInputApplied + 1
```

(a)

```
21: procedure FILLOUTPUTMESSAGES(outputMsg)
22:     if (started) then
23:         outputMsg.add("Execute Water", True)
24:     else if (isWaterExecuted) then
25:         outputMsg.add("Get Water Data", True)
26:         isWaterExecuted ← False
27:     else if (isEnergyExecuted) then
28:         outputMsg.add("Get Energy Data", True)
29:         isEnergyExecuted ← False
30:     else if (WaterInputApplied == 1) then
31:         outputMsg.add("Execute Water", True)
32:         WaterInputApplied ← 0
33:     else if (EnergyInputApplied == 2) then
34:         outputMsg.add("Execute Energy", True)
35:         EnergyInputApplied ← 0
```

(b)

```
36: procedure FILLOUTPUTMESSAGES(outputMsg)
37:     if (started) then
38:         outputMsg.add("Execute Water", True)
39:         outputMsg.add("Execute Energy", True)
40:     else if ((isWaterExecuted) & (isEnergyExecuted)) then
41:         outputMsg.add("Get Water Data", True)
42:         outputMsg.add("Get Energy Data", True)
43:         isWaterExecuted ← False
44:         isEnergyExecuted ← False
45:     else if ((WaterInputApplied == 1) & (EnergyInputApplied == 2)) then
46:         outputMsg.add("Execute Water", True)
47:         outputMsg.add("Execute Energy", True)
48:         WaterInputApplied ← 0
49:         EnergyInputApplied ← 0
```

(c)

Figure 5.23: The Sketch of the Defined Execution Control for the WEN Example. (a) The Perform and CheckInputMessage Procedures. (b) FillOutputMessages to Apply Sequential Execution. (c) FillOutputMessages to Apply Parallel Execution.

5.3.8 Model Validation

Verification is the process of determining that a model implementation and its associated data accurately represent the developer's conceptual description and specifications (answering the question "Have we built the model, right?"). Validation is the process of determining the degree to which a simulation model and its associated

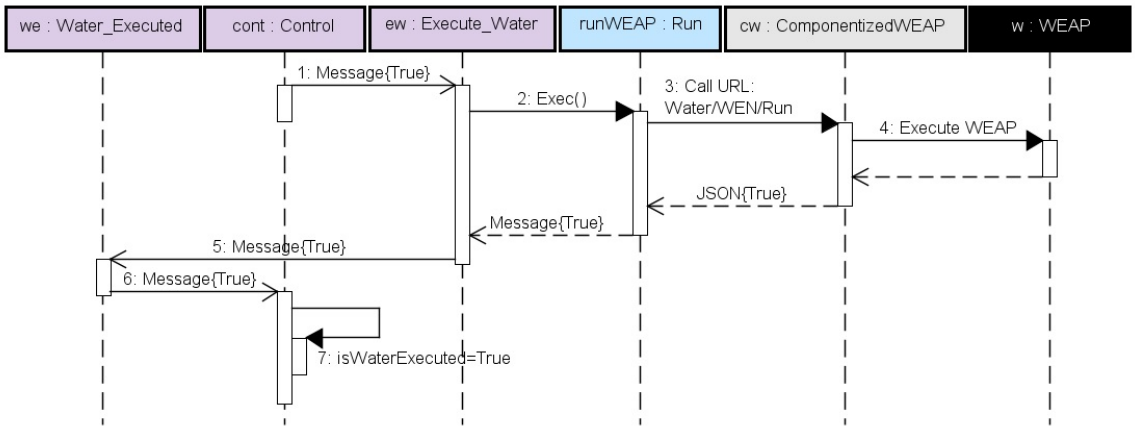


Figure 5.24: Sequence Diagram to Execute the Water Model (Defined Model in the WEAP System) by a DEVS-IM Model.

data accurately represent the real world from the perspective of the intended uses of the model (answering the question "Have we built the right model?").

Figure 5.25 illustrates the parts and their relationships for the internal linkage and interaction model. The *Read(x)* on the arrows between the systems means reading the x variable from the source system. Also, the *Write(x,y)* means writing the values of the variable x (from the source system) to the y variable (from the target system). The WV and LV in the formulas represent the "WEAP Variables" and "LEAP Variables", respectively. In the WEAP-LEAP internal linkage (see Figure 5.25a), the value of a WEAP/LEAP variable drives from some function(s) of the WEAP variables, some function(s) of the LEAP variable, and some constants. As an example, the values of the WV_i calculated by reading the LV_p variable from the LEAP system, reading the WV_o variable from the WEAP system, and constant-coefficient values (i.e., c_1, c_2, and c_3).

In the Interaction Model approach (Figure 5.25b), the data transformation formulas can be completely or partially applied to the data in the interaction model. For example, the computation formulas for the WV_i in Figure 5.25b applied completely, and the computation formulas for the WV_j applied partially to the data in

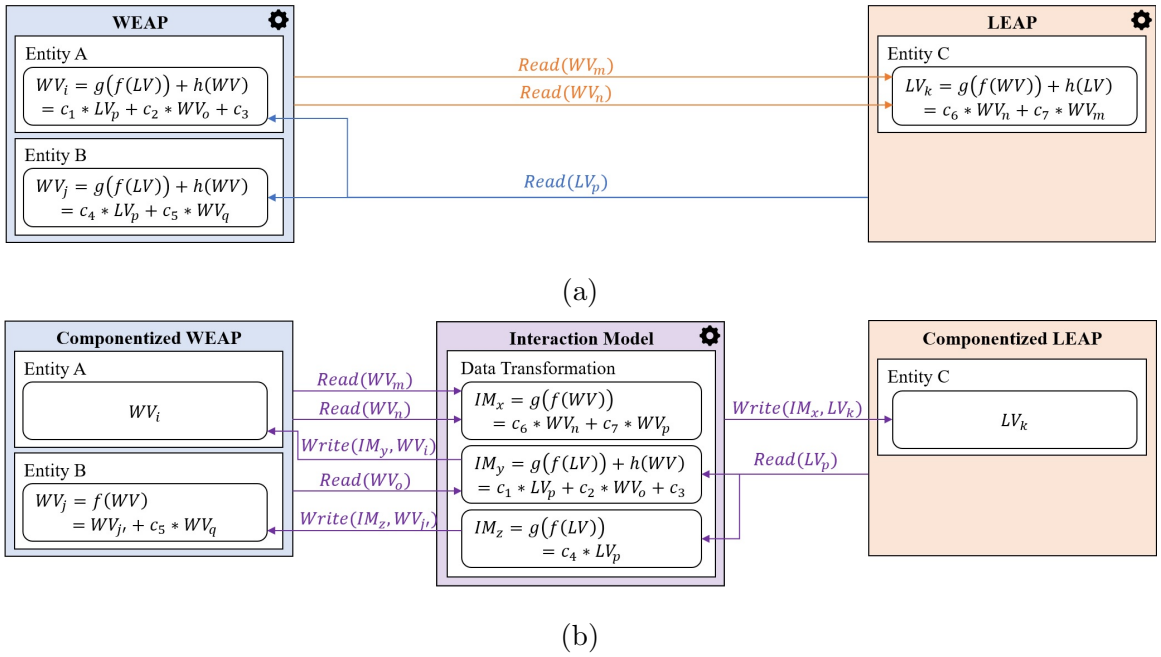


Figure 5.25: Comparing the Data Processing in the WEAP-LEAP Internal Linkage and Interaction Model.

the Interaction Model. Completely applying data transformation to the Interaction Model formalizes and reduces the complexity of the relationship between the WEAP and LEAP models. In the WEAP-LEAP internal linkage, the systems are responsible for reading the required data from the other system and applying the data transformation before starting the execution (the *Read(x)* on the arrows between the system in Figure 5.25a). Whereas, in our approach, the Interaction Model is responsible for reading the data from the system, performing data transformations, writing the transformed data to the system, and controlling the execution of the componentized WEAP and LEAP systems (i.e., managing the *Read(x)* and *Write(x,y)* operations for time-driven data transformations shown in Figure 5.25b).

A variety of methods are used to validate simulation models; for example, "Face Validity", "Historical Data Validation", "Sensitivity Analysis", an so on. (Law, 2019). The validation in this research is based on the "Comparison to Other Models" method. The WEAP-LEAP internal linkage model is considered to be the real-world data for

validating the DEVS-IM model. The initial states of the "Phoenix AMA" model in the WEAP and LEAP systems are the same as those in the WEAP-LEAP internal linkage and DEVS-IM models. The water and energy "Phoenix AMA" systems are defined as deterministic mass-balance equations. Given the same WEAP and LEAP models with the same initial states, the transformation formulas defined in the Interaction Model should have the same results as those of the internal linkage.

The results of executing the Phoenix AMA DEVS-IM model are validated in two scenarios. First, the WEAP model is executed (considering zeros for the dependent variables to the LEAP system), its results are read, and the transformed data are applied to the LEAP model. Then, the LEAP system is executed (using the applied data by the Interaction Model), its results are read, and the transformed data are applied to the WEAP model. At this step, the results of both WEAP-LEAP internal linkage and DEVS-IM approaches are exported to CSV files. Then, a separate application (written in Java programming language) compares the results based on the components, variables, years, and time steps. The outcome shows that the results are identical in most cases. In some cases, the results have negligible differences ($\sim 10^{-6}$) due to the transferred data precision and computation/rounding mechanism. The WEAP and LEAP APIs allow extracting the data from their systems up to 15 digits (maximum six decimal places). However, their computation engines may use data with higher precision. Second, the same mechanism, but first running the LEAP system and then the WEAP system, is applied. Consequently, simulating the WEN model via the WEAP-LEAP internal linkage and the DEVS-IM have nearly identical results.

Table 5.2 compares the main differences in modeling and simulation an interaction model using the Algorithmic-IM or DEVS-IM frameworks.

Table 5.2: Main Differences in Modeling and Simulation an Interaction Model Using the Algorithmic-IM and DEVS-IM Frameworks.

Algorithmic-IM	**DEVS-IM**
Designed based on the software design principles	Designed based on the system theory and formal specification
Having a flat modeling structure (one level of hierarchy)	Having a component-based structure for modeling (hierarchical)
Concatenated model specification and simulation protocol	Separate Model specification and simulation protocol (based on the DEVS formalism)
Fixed execution control	Dynamic step-by-step execution control
Storing the models in flat files (java classes)	Storing the models in MongoDB database
Defining the Model via Programming Language	Defining the Model via REST APIs
Direct communication between the interaction model and disparate systems	Communication between the interaction model and disparate systems via a separate interface for each external system
Simpler to use, but restricted (in both modeling and simulation)	More complex to use, but more flexible and maintainable (in both modeling and simulation)

Chapter 6

CASE STUDY
(WATER-ENERGY NEXUS)

The metropolitan Phoenix, a city in the US desert southwest, is a subject study area for understanding the water and energy nexus. As an Active Management Area (AMA) region, sustainable aquifer withdrawal/recharge is one of the primary goals to be achieved by 2025 (Guan et al., 2020). Both water and energy serve as both demand and supply. They form a supplier and consumer collective where they control one another. There are other players that directly or indirectly affect/affected by the water-energy feedforward and feedback relationships. In the context of this research, a study of the water and energy system for the Phoenix AMA is carried out using the WEAP and LEAP systems (Guan et al., 2020; Mounir et al., 2019). To develop and calibrate water and energy models for the Phoenix AMA, the WEAP and LEAP systems, and publicly available data sources. Data sets from 1985 through 2009 are used to develop the water and energy models. Simulation scenarios cover the period starting in 2008 and ending in 2018.

An illustration of the water and energy entities for the Phoenix AMA is shown in Figure 6.1. This visualization of the water-energy system provides a simplified representation of all the Phoenix AMA entities that are developed and validated. The water system is supplied by four primary sources: the Salt River Project (SRP), the Central Arizona Project (CAP), groundwater, and reclaimed water. Also, the main water demands in the area are for irrigation, municipal, industry, Indian communities, riparian, and energy generation (power plants) (Guan et al., 2020). The water needs to be transmitted from the supply sources to the demand nodes. This high-fidelity

water model has two *River* (SRP and CAP), two Groundwater, and two *Wastewater Treatment Plant* entities, collectively representing the water supply. This model has twelve *Catchment* (for irrigation) and six *Demand Site* (for the rest of demands) entities, representing the water demand. The water supply and demand entities form a network using fifty-six *Transmission Link* and five *Return Flow* entities.

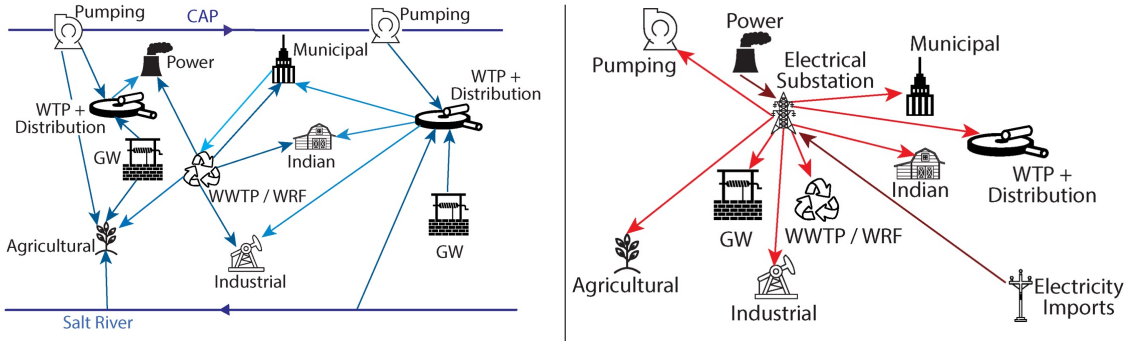


Figure 6.1: An Illustration of a Model for the Phoenix AMA Water-Energy System.

The energy model for the Phoenix AMA has the goal of tracking the energy embedded in all water uses and infrastructures (Mounir et al., 2019). The energy model is supplied by two utilities: Salt River Project (SRP) and Arizona Public Service Company (APS). They mainly use Coal, Nuclear, Natural Gas, and Renewable (solar and wind) primary resources to generate electricity. The energy is supplied via two entities: one for electricity generation and another for electricity transmission and distribution. All customer electricity demand is provided by APS, SRP, and CAP. The energy demands are defined via three main sectors: Residential, Commercial, and Industrial. The first two sectors are affected by population growth (e.g., required energy for water heating), but the last sector is for satisfying the water-related energy demands required for pumping, water treatment, and distribution. The Industrial sector also has a subsector for water treatment facilities (Wastewater Treatment Plant (WWTP) and Water Reclamation Facilities (WRF)). The energy model has nine *Re-*

source, one *Transformation* with twenty-seven *processes*, and ninety-three *Demand* entities.

Sustainable aquifer withdrawal/recharge is one of the primary goals to be achieved in the Phoenix AMA by 2025 (Guan et al., 2020). A study of the water and energy system for the "Phoenix AMA" is carried out using the WEAP and LEAP systems. The food model is defined inside the water model (using the WEAP-MABIA). The defined water model for the "Phoenix AMA" in the WEAP system has six water supplies (two *River*, two *Groundwater*, and two *Wastewater Treatment Plant* entities) and 18 demands (12 *Catchment* for irrigation and six *Demand Site* entities for the rest of the demands) using 61 connections between them (56 *Transmission Link* and five *Return Flow* entities) to form the water network. The defined energy model for the "Phoenix AMA" in the LEAP system has nine *Resource*, one *Transformation* with 27 *Processes*, and 93 *Demand* entities. The water and energy models have 183 (172 from water to energy and nine from energy to water) interactions with each other. The whole WEN model for the "Phoenix AMA" has been developed using the WEAP-LEAP internal linkage (Guan et al., 2020; Mounir et al., 2019), Algorithmic-IM framework (Fard et al., 2020), and DEVS-IM framework (Fard & Sarjoughian, 2022). The results of the DEVS-IM framework contribute to the transparency of building high-fidelity simulations of water-energy systems (compared to the data-sharing mechanism used by the WEAP-LEAP internal linkage).

6.1 WEN Modeling for the Phoenix AMA via WEAP-LEAP Internal Linkage

As mentioned in Section 3.3, the WEAP and LEAP systems have an internal linking mechanism that can bi-directionally share data to read variables from one to another (given satisfying the restrictions). An illustration of a portion of the water and energy models for the "Phoenix AMA" is shown in Figure 6.2. A portion of the

schematic of the water model and the Data views (tree structures) for the water and energy models are shown, as well. The connections between the water and energy model entities are hand-drawn as solid and dotted lines. The dotted blue lines from the water model to the energy model illustrate the flow from the transmission link entities to the demand entities. The calculations for the amount of energy that can be produced given the amount of water that can be made available are defined in the energy entities (e.g., Kyrene Generating Station). The solid red lines illustrate the flow of power (electricity) from the energy model to the water model. Similarly, the amount of water that can be supplied to meet demand depends on the amount of energy. These calculations are defined in the appropriate water entities (e.g., demand site). The "Power Plant" demand site entity in the water model needs to know the amount of generated electricity by nine processes in the "Electricity Generation" transformation entity in the energy model (to fill the "Monthly Demand" variable). Also, the "Treatment and Distribution" demand entity in the energy model needs to know the amount of flow in the transmission links from "Groundwater" and "GW-Backup" to the "Municipal" in the water model (to fill the "Final Energy Intensity" variable). Calculations for aggregating and/or converting the generated electricity in the energy model must be defined in the "Power Plant" demand site entity in the water model. Similarly, calculations for aggregating and/or converting the amount of flow in the water model must be defined in the "Treatment and Distribution" demand entity in the energy system.

6.2 WEN Modeling for the Phoenix AMA via Algorithmic-IM

The Componentized WEAP and Componentized LEAP systems, the encapsulated WEAP and LEAP systems in RESTful frameworks, are coupled using an Interaction Model (Fard & Sarjoughian, 2020). The coupling follows the Knowledge Interchange

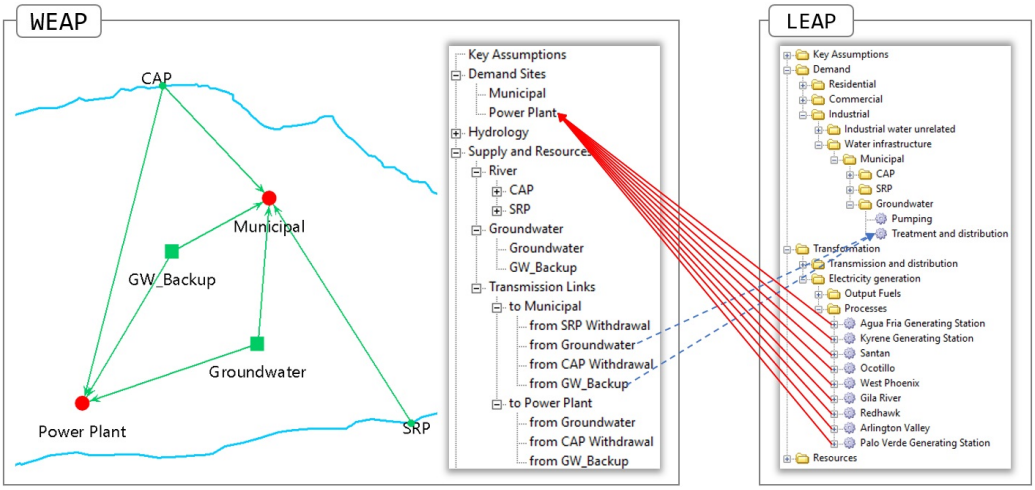


Figure 6.2: A Portion of the Water Model Schematic and Data View, Energy Model Data View, the Water-Energy Model, and Illustrated Connections Linking the Water and Energy Models for the Phoenix AMA.

Broker (KIB) approach, where relationships between the two disparate models are defined as separate models (Sarjoughian, 2006). A portion of the internally linked water and energy model (see Figure 6.2) is depicted in a component view in Figure 6.3. This diagram illustrates the logical specifications for the water and energy model interactions. An interaction model has a set of modules, each responsible for transforming data from one model for use by another model. Each module has its own input/output ports to receive/send the data from/to water and energy models. The module's ports are connected to specific entities and variables in the Componentized WEAP and Componentized LEAP models. The structure of the received or sent data in the module's ports must be defined in the interaction model. Each module has one-to-many transformations, where each transformation has its input and output ports. Three types of coupling are allowed between the module and/or transformation ports. They are modules' input to transformations' input, transformations' output to transformations' input, and transformations' output to modules' output (see Figure 6.3). Each transformation can process one or more data values on its input ports and send data values on its output ports. The interaction model has

a cyclic, time-step, synchronous execution protocol for concurrent and bidirectional data transformations between water and energy models. With the use of the interaction model, the water and models do not need to have direct knowledge of each other. The WEAP model has the data (input and output) of the type Flow, and the LEAP model has the data (input and output) of the type Electricity. Since the water and energy models are decoupled, they can execute concurrently with one another and the interaction model in every simulation cycle.

The interaction model is responsible for receiving, processing, and sending the data required for the water and energy models, as depicted in Figure 6.3. The Treatment and Distribution demand component in the energy model does not have any knowledge about the water source. It receives the required generated electricity from the module named "Mun-GW.treat" port of the interaction model. In this example, the "Treatment and Distribution" demand component in the energy model and the "Groundwater to Municipal" and "GW-Backup to Municipal" transmission components in the water model are coupled using the transformation "F-E" (Flow to Electricity) in the module named "Mun-GW". The processes in the "Electricity Generation" transformation component in the energy model and the "Power Plant" demand site component in the water model are coupled using the transformation "E-F" (Electricity to Flow) in the module named "Elect-Flow". The port names defined for the modules and transformations in the Interaction Model are provided in Figure 6.3.

The IM communicates with the Componentized WEAP/LEAP using the designated APIs. According to the signature of the APIs, every module's ports need to know the *project name*, *component type*, *component name*, *variable type*, *variable name*, and *scenario name* of the componentized water and/or energy models. The structure of the incoming/outgoing messages to/from the interaction model is defined

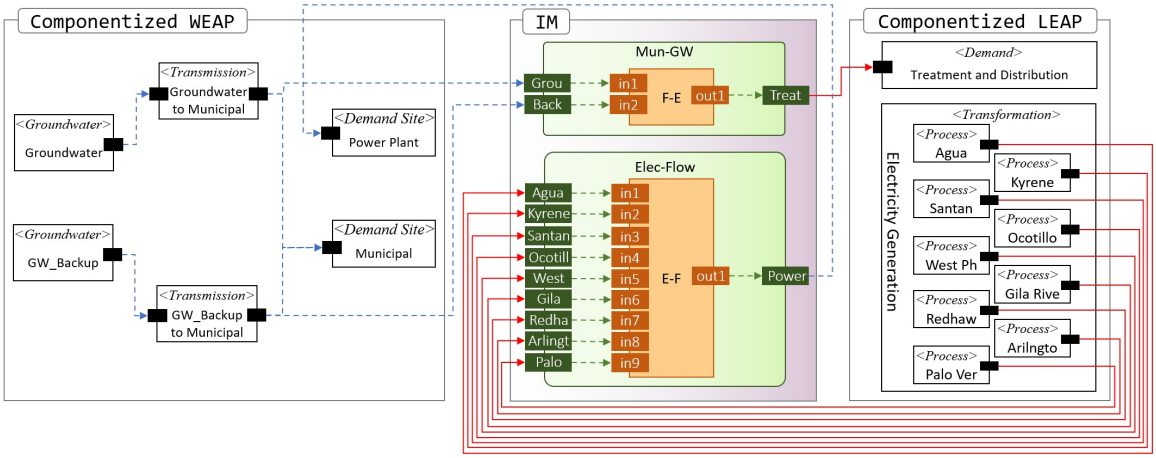


Figure 6.3: An Illustration of the Algorithmic-IM for a Portion of the Phoenix AMA WEN Model Using Componentized WEAP and LEAP.

according to the APIs of the Componentized WEAP/LEAP RESTful framework. In the AMA interaction model, the data from the water and energy models (*WEAPMessage* and *LEAPMessage*) share the same structure. Each message has a finite number of time intervals; each time interval has a year value and a finite number of data; each data has a time-step/time-slice and a value.

In the Phoenix AMA model, the demand for the energy model needs the values of the *Flow* result variable of two transmission links in the water model. These values (with m^3 unit) are converted to values (with KW/h unit) for the *Energy Intensity* variable of a demand entity in the energy model using a coefficient. Essentially, the "F-E" transformation defines data conversion from "Groundwater to Municipal" and "GW-Backup to Municipal" transmission entities (in the Componentized WEAP model) to the "Treatment and Distribution" demand entity (in the Componentized LEAP model). Similarly, the electric data from the energy model is read and converted to water quantity for the water model (in the "E-F" transformation of the IM). The required computation in the interaction model handles by the transformations. The data conversion calculation for the Phoenix AMA interaction model is shown in Equation 6.1. Transformation's input ports are read (based on the year

and time-step/time-slice) and multiplied by a factor F to calculate the output values. The range of the year and time granularity (time-steps in the WEAP model and time-slices in the LEAP model) parameters are assigned by the corresponding values in the source model of the transformation.

$$Out.v_{y,ts} = \sum_{in \in tran.inputs} [in.v(y, ts) \times F_{in,y,ts}] \tag{6.1}$$

$$y \in N, startYear \leq y \leq endYear; ts \in N, 1 \leq ts \leq \#timeGranularityForYear$$

The factor F in Equation 6.1 can be constant for all input values or specified for each input port, year, and time-step/time-slice. For example, F is $130.34 \times 0.000810714$ for all the input ports, years, and time-steps in the "F-E transformation" in Figure 6.3. For the "E-F transformation" in Figure 6.3, F is different based on the input ports, years, and time granularities (months). Equation 6.2 shows the computation in the "E-F transformation" in Figure 6.3. The value for each k is predefined to be the number of days d for a month in each year multiplied by 24. For example, the value of k is 31×24 for Jan. 2018. The simulation duration is 11 years ($2008 \leq y \leq 2018$) with monthly time granularity ($1 \leq ts \leq 12$). The charts in Figure 6.4 show the input and output port values of the transformations (i.e., "E-F" and "F-E") for 2018.

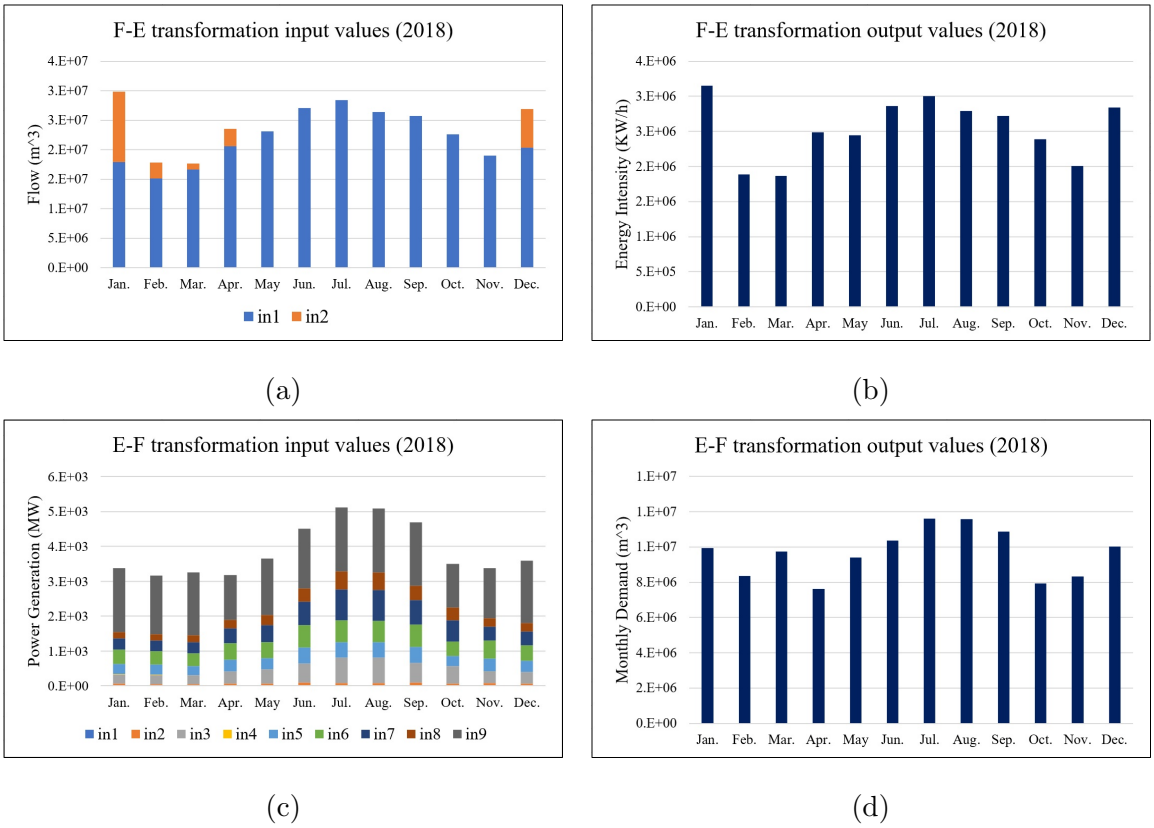


Figure 6.4: The Input and Output Data of the Interaction Model Simulation for the Phoenix AMA Area. (a) The "F-E" Transformation Inputs, (b) The "F-E" Transformation Output, (c) The "E-F" Transformation Inputs, and (d) The "E-F" Transformation Output.

$$
\begin{aligned}
Out1.value_{y,ts} = {} & in1.v(y,ts) \times F_{in1} + in2.v(y,ts) \times F_{in2} \\
& + in3.v(y,ts) \times F_{in3} + in4.v(y,ts) \times F_{in4} \\
& + in5.v(y,ts) \times \left\{ \begin{array}{ll} F_{in51} & y \leq 2013 \\ F_{in52} & otherwise \end{array} \right\} \\
& + in6.v(y,ts) \times \left\{ \begin{array}{ll} F_{in61} & y \leq 2013 \\ F_{in62} & 2013 < y \leq 2013 \\ F_{in63} & otherwise \end{array} \right\} \\
& + in7.v(y,ts) \times F_{in7} + in8.v(y,ts) \times F_{in8} + in9.v(y,ts) \times F_{in9}
\end{aligned}
\tag{6.2}
$$

$$F_{in1} = 1.9 \times k; F_{in2} = 3.1 \times k; F_{in3} = 1.3 \times k; F_{in4} = 4.4 \times k; F_{in51} = 3.2 \times k; F_{in52} = 1.6 \times k;$$

$$F_{in61} = 7 \times k; F_{in62} = 4 \times k; F_{in7} = 1.08 \times k; F_{in9} = 6.006 \times k;$$

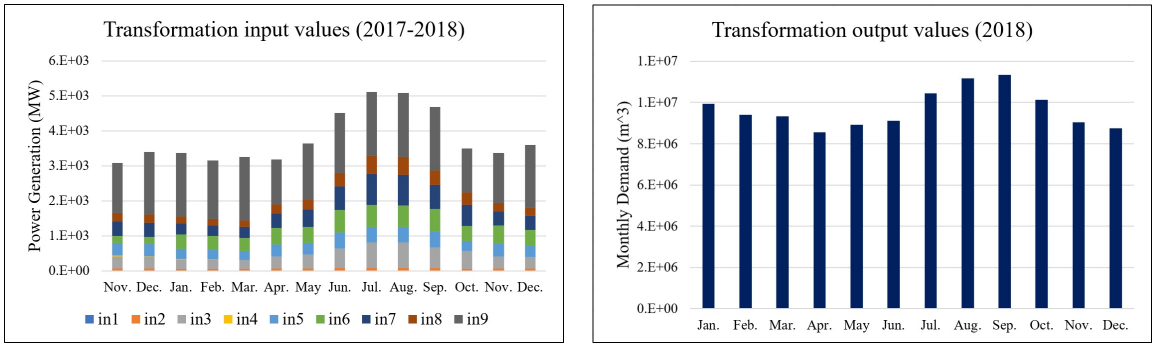
Specifying transformations depends on the specifics of the water and energy models. However, using interaction modeling, data transformations can be defined as separate dynamical models instead of making changes to the water and energy models. For example, Equation 6.3 is a hypothetical data conversion calculation from the energy output to the water input. The transformation's input ports are read (based on the year and time-step) and multiplied by a factor F. The output value for a given year and time-step/time-slice is the average of the previous p time steps in the source model of the transformation.

$$Out.v_{y,ts} = \sum_{in \in tran.inputs} \frac{\sum_{i=ts-p}^{ts} \begin{cases} in.v(y-1, TG+i) \times F_{in,y-1,TG+i} & i \leq 1 \\ in.v(y,i) \times F_{in,y,i} & i > 1 \end{cases}}{p} \quad (6.3)$$

$$y \in N, startYear \leq y \leq endYear; ts \in N, 1 \leq ts \leq \#timeGranularityForYear(TG); p \in N$$

The charts in Figure 6.5 show the input and output port values of a hypothetical transformation for the Phoenix AMA interaction model. Each month's output value is the average of the current and its previous two months in the LEAP model (see Equation 6.3). For example, the output for Feb. 2018 is the average of the Feb. 2018, Jan. 2018, and Dec. 2017 inputs values. As expected, the outputs in Figure 6.4d and Figure 6.5b are two different interactions between the water and energy models, representing two different water-energy nexuses. This example in Figure 6.5 shows one of many alternative ways the nexus dynamics of the water-energy model could have regulated the data exchanges between the energy and water models. In a standalone fashion, the interaction model serves to explore different nexus policies that can satisfy combined supply and demands both within and between water and energy systems.

The schemes of the complete Phoenix AMA nexus model via the WEAP-LEAP internal linkage and the interaction model are shown in Figure 6.6. The blue and



(a) (b)

Figure 6.5: The Input and Output Data of the Interaction Model Simulation for the Phoenix AMA Area. (a) The Transformation Inputs from Nov. 2017 to Dec. 2018, (b) The Transformation Output for 2018.

red arrows between the componentized water and energy models and the interaction model represent the *Flow* and *Electricity* data of the source entity (or component), respectively. The water and energy model components are depicted as boxes with black borders and include their ports as filled black boxes without names. The ports located on the left (right) side of every component are designated as inputs (outputs). In Figure 6.6a, the "Municipal" demands in the energy model read the *Flow* result variables of the corresponding transmission links of the water mode. The "SRP" and "CAP" demands need to read the *Flow* of one transmission link, whereas the groundwater demands need to read the *Flow* of two transmission links. In Figure 6.6b, the interaction model illustrates receiving the *Flow* output from the transmission components of the Componentized WEAP model (i.e., the *WEAPMessage*), transforming the data based on the time-step and/or time-slice, and sending the transformed data (i.e., the *LEAPMessage*) to the corresponding demand components of the Componentized LEAP model. The "Indian" and "Industrial" demands are the same as the "Municipal" demands. Figure 6.6c shows the WEAP-LEAP internal linkage, and Figure 6.6d illustrates the interaction model for the "Agricultural" demands. Each "CAP", "SRP", and "Groundwater" demand in the LEAP model needs to read the

Flow result variable of 8, 10, and 12 different transmission links, respectively. Modules and transformations in the interaction model have the same functionality as described for the "Municipal" demands, except the module "Agr-CAP" which has many transformation's output ports. All the conversions with one output and multiple inputs in the Phoenix AMA interaction model in Figure 6.6 are calculated using a formula with different configurations. Interaction modeling provides flexibility to make changes to each module independently of other modules as well as decomposing complex interactions into simpler ones in a systematic fashion.

To better understand these different approaches of the water-energy nexus modeling described in the previous sections, the stages in their execution cycles, as shown in Figure 42, can be examined in detail. The execution steps show holistic computation time periods for simulating the Phoenix AMA water and energy models and their nexus. This model has a total of 85 WEAP entities and 103 LEAP entities and 13 modules, and 82 transformations in the Interaction model.

Considering the models shown in Figure 37 and Figure 6.7a and 6.7c, it can be seen every simulation cycle is comprised of computations for executing the entities of the water model and all the entities of the energy model. Figure 6.7a shows the computation time for every cycle that includes reading the data needed for executing the water and energy models. The total time for an execution cycle for the internal linkage approach is $\delta t_w + \delta t_l$. The time periods for the water and energy model to read data from one another are $\delta t_w, 0 < \delta t_{wr} < \delta t_w$ and $\delta t_{lr}, 0 < \delta t_{lr} < \delta t_l$. The computation times for the WEAP-LEAP internal linkage is 169.5 seconds, with 25% and 75% consumed by the WEAP and LEAP systems (see Figure 6.7b).

The breakdown of the computation time for the interaction models is shown in Figure 6.7c. The computation time for a complete execution cycle is the maximum of the δt_w for executing the water and δt_l for the energy models. The computation times

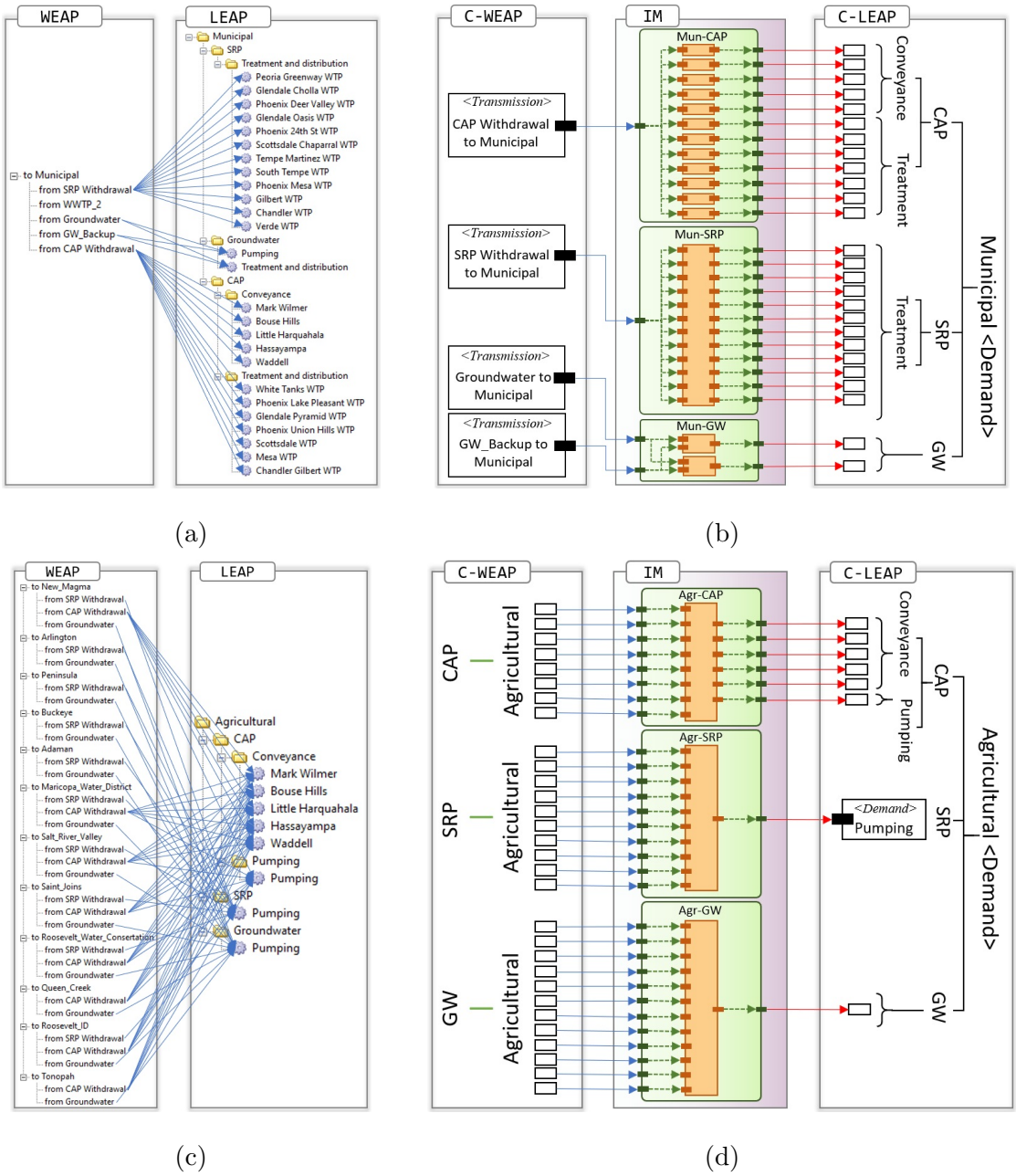


Figure 6.6: Comparing the WEAP-LEAP Internal Linkage and the Algorithmic-IM for the Phoenix AMA Model. (a) the Internal Linkage for the Required Electricity by the "Municipal" Demands. (b) The Interaction Model for the Required Electricity by the "Municipal" Demands. (c) The Internal Linkage for the Required Electricity by the "Agricultural" Demands. (d) The Interaction Model for the Required Electricity by the "Agricultural" Demands.

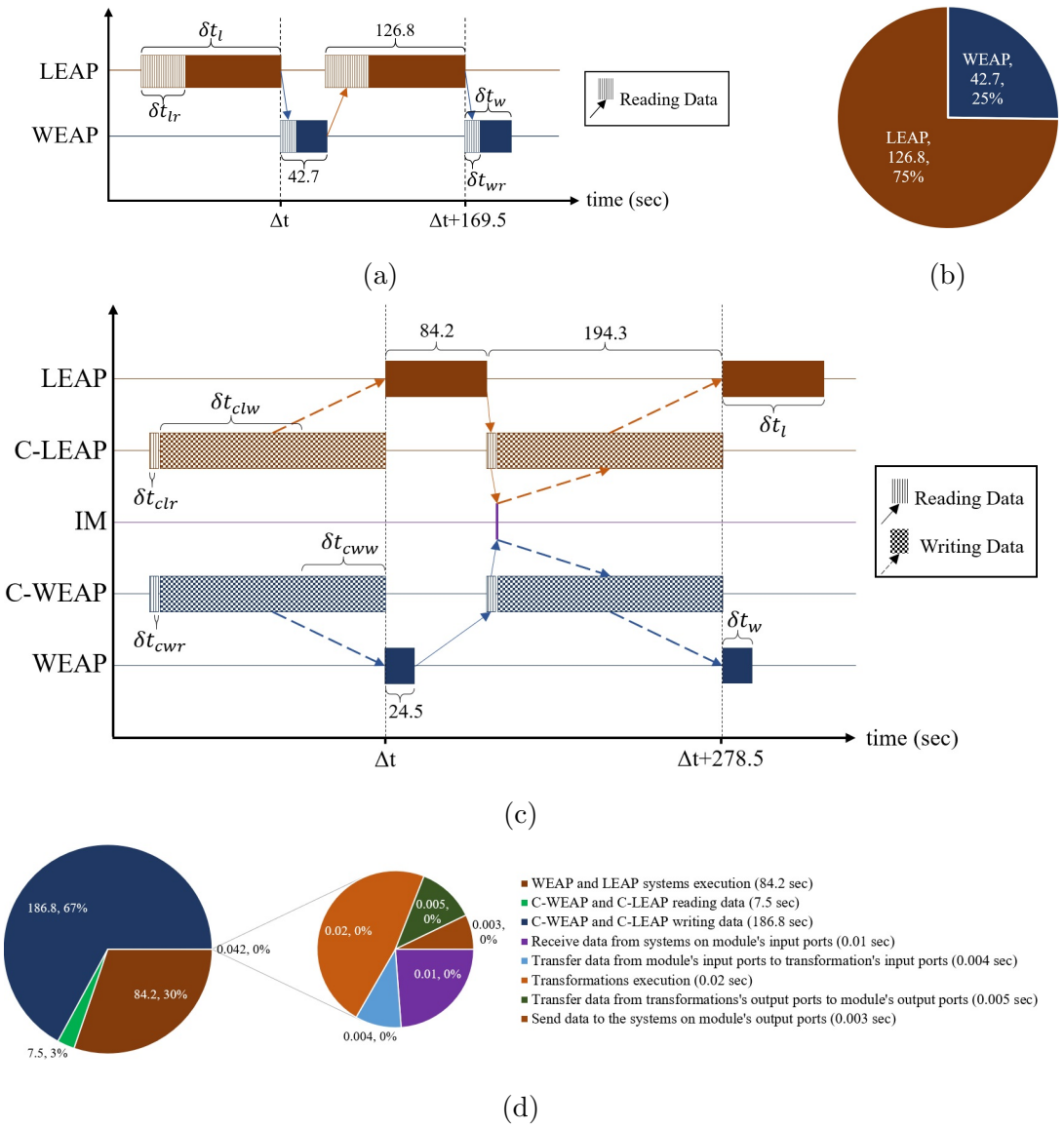


Figure 6.7: Stages for Modeling the Phoenix AMA Nexus Using the WEAP-LEAP Internal Linkage and Algorithmic-IM. (a) Execution Periods for the WEAP-LEAP Internal Linkage. (b) Execution Time Percentages for the WEAP-LEAP Internal Linkage. (c) Execution Periods for the Algorithmic-IM. (d) Execution Time Percentages for the Algorithmic-IM.

for the WEAP and LEAP systems are 24.5 and 84.2 seconds, respectively. The APIs for the Componentized WEAP and Componentized LEAP frameworks, executed in parallel on multiple CPU cores, consume time for receiving and reading and data (i.e., $\delta t_{cwr} + \delta t_{clr}$) from the WEAP and LEAP systems. The $\delta t_{cwr} + \delta t_{clr}$ period is shown

for the Componentized WEAP and Componentized LEAP parts of the execution cycle. The $\frac{\delta t_{cwr}}{\delta t_{cwr}+\delta t_{clr}}$ portion of the $\delta t_{cwr}+\delta t_{clr}$ section belongs to the Componentized WEAP framework, and the rest belongs to the Componentized LEAP framework. The portion of the execution cycle time for writing and sending data to the WEAP and LEAP systems is $\delta t_{cwwr}+\delta t_{clw}$. As shown in Figure 6.7c, the IM consumes a small portion of time for all the computations in the IM modules. Figure 6.7d shows the breakdown of periods consumed for executing the interaction model. The periods for $\delta t_{cwr}+\delta t_{clr}$ and $\delta t_{cww}+\delta t_{clw}$ are 7.5 and 186.8 seconds, respectively. The period for transformations in the modules is 42 milliseconds. The periods allocated to different tasks in the Interaction Model are shown in Figure 6.7d. The execution time for the interaction model is 64% higher than the time needed for the internal linkage. The Phoenix AMA model was simulated ten times using the internal linkage and ten times using the interaction modeling approaches. A standalone desktop computer with Windows-10 64-bit OS with four Core i5 Intel CPUs and 20 GB RAM is used for all the experimental results.

6.3 WEN Modeling for the Phoenix AMA via DEVS-IM

The DEVS-IM REST APIs are used to define the structure of the Phoenix AMA Water-Energy model (via around 2,000 APIs). The model has two interfaces for the entities belonging to the Componentized WEAP and Componentized LEAP systems. The interfaces are defined using 181 *Component* and 144 *Function* elements (see Section 5.3.2). The interaction model has 15 *Process*, 91 *Task*, and 201 *Connector* elements. The interaction model has around 307 atomic and coupled DEVS models with 942 ports and 740 couplings. This interaction model replicates the same WEN model developed using the WEAP-LEAP internal linkage (same as the devel-

oped model using the Algorithmic-IM framework). The DEVS-IM model structure is verified for correctness before storing it in MongoDB.

Using the code generator module, the skeleton of the DEVS-Suite simulator project is created using the stored data in the MongoDB database. A package with the same name as the DEVS-IM project (called "Project") is added under the `InteractionModel` package (which is under the `Models` package). The root of the interaction model (a coupled DEVS model) is implemented using Java with the same name as the DEVS-IM project and a package (DEVS-IM project's name + "Models") for the sub-models (i.e., "Project.java" file and "ProjectModels" package). The same approach is applied to implement the *Process* elements of the interaction model (i.e., "Coupled.java" file and "CoupledModels" package). Also, all *Task* elements of the interaction model are implemented using Java (i.e., "Atomic.java" file). This approach is used for the whole hierarchy of the interaction model. Likewise, each external system is implemented using Java with the same name as the external system and a package (external system's name + "Components") for its sub-components (i.e., "System.java" file and "SystemComponents" package). Each component is implemented using Java and two packages for its sub-components and functions (i.e., "Entity.java" file and "Entity-Components" and "EntityFunctions" packages). This approach is also used to define an interface for the WEAP or LEAP systems. Figure 6.8 presents a portion of the generated files and packages for the DEVS-IM model in the DEVS-Suite simulator. The "PhoenixAMA.java" file and "PhoenixAMAModels" package contain the required files to model the interaction model. Also, the "WEAP.java" and "LEAP.java" files and "WEAPComponents" and "LEAPComponents" packages contain the required files to define the interfaces for the WEAP and LEAP systems.

Having an interface for the external system increases the model complexity, but it has two main advantages. First, it promotes the interaction model to define the

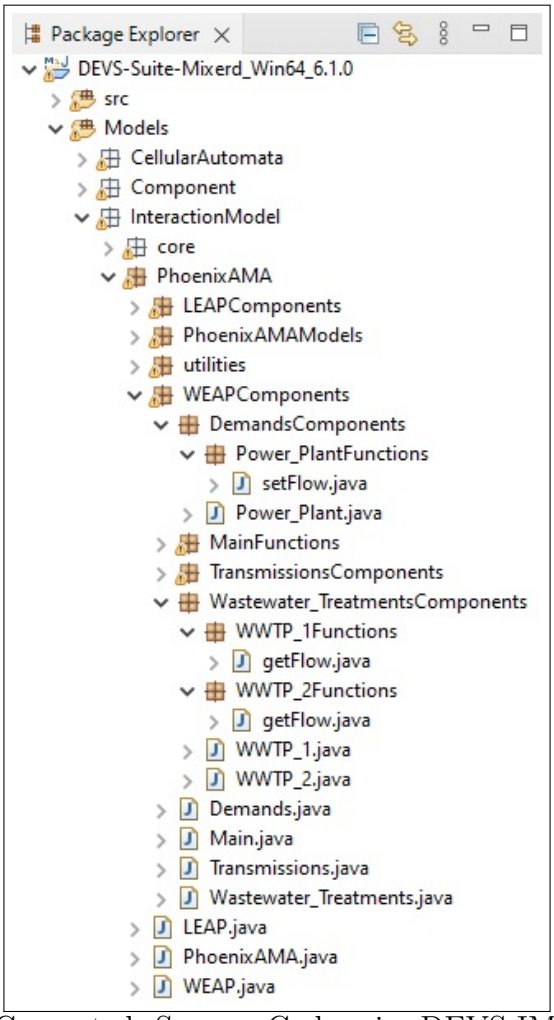


Figure 6.8: The Generated Source Code via DEVS-IM Framework for the "PhoenixAMA" Project in the Eclipse IDE.

external system at different levels of abstraction or ignore some parts of the actual model. Second, it helps to have a high level of modularity between how to get data from the external systems and how the interaction model is manipulating the sent/received data. As an example, the "Phoenix AMA" WEAP model has six *Demand Site* entities, but just one of them is used inside the "Phoenix AMA" DEVS-IM model (presented via `Models/InteractionModel/PhoenixAMA/WEAPComponents/DemandsComponents/Power_Plant.java` in Figure 6.8).

Figure 6.9 shows a portion of the "Phoenix AMA" DEVS-IM model in the DEVS-Suite simulator's SimView which hides the sub-models of some coupled models (presented in black-box mode). For example, the sub-models of the "Municipal_CAP" and "Municipal_SRP" coupled models are hidden in Figure 6.9. The black-box mode in the DEVS-Suite simulator allows hierarchical viewing of large-scale models. The white-box mode (shown for "Municipal_Groundwater" coupled model in Figure 6.9) presents one level of the hierarchy for the coupled models. This viewing mode is for developing and debugging complex models via step-by-step tracking of the input and output messages among models and monitoring the states of atomic models.

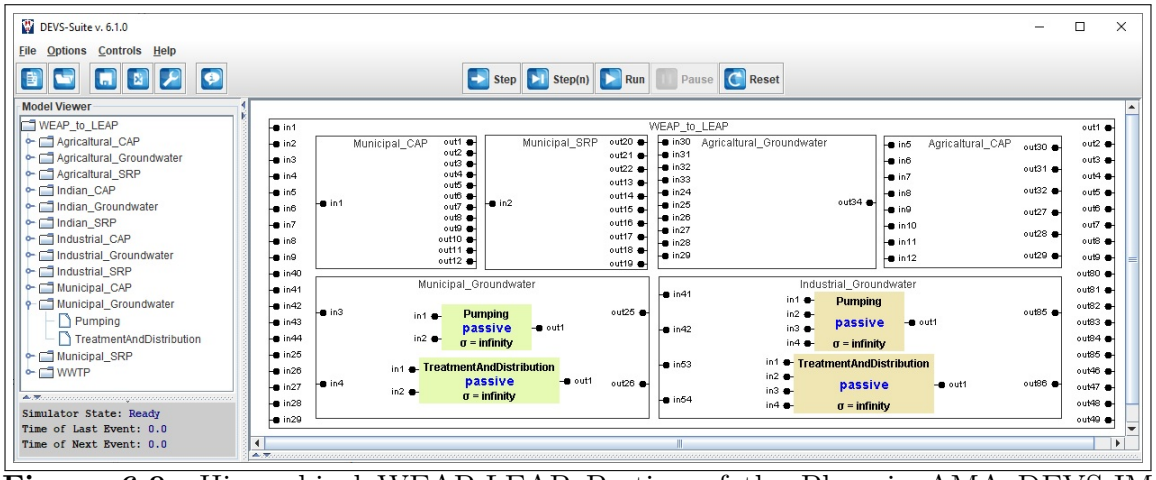


Figure 6.9: Hierarchical WEAP-LEAP Portion of the Phoenix AMA DEVS-IM Model Depicted in the DEVS-Suite Simulator's SimView.

Figure 6.10 presents another portion of the DEVS-IM model in the DEVS-Suite simulator. The output connectors are connected to the Componentized WEAP and Componentized LEAP systems. The purple models are used to control the execution, and the gray models are used for time-based data transformations. Applying the data transformation to the received data from the external systems must be defined by the modeler via adding behavior to the Task elements (the atomic models in the DEVS-Suite simulator). Also, the execution control for the whole interaction model must be defined in a Task element named "Control". The "Control" model defines the ordering

of receiving/sending data from/to the external systems and the order of executing the WEAP and LEAP systems. As shown in Figure 6.10, the "Control" model has five inputs ("start", "LEAP Executed", "LEAP Input Applied", "WEAP Executed", and "WEAP Input Applied") and four outputs ("Get LEAP Data", "Get WEAP Data", "Run LEAP", and "Run WEAP"). The "Control" model replicates the execution of the WEAP and LEAP systems via the WEAP-LEAP internal linkage.

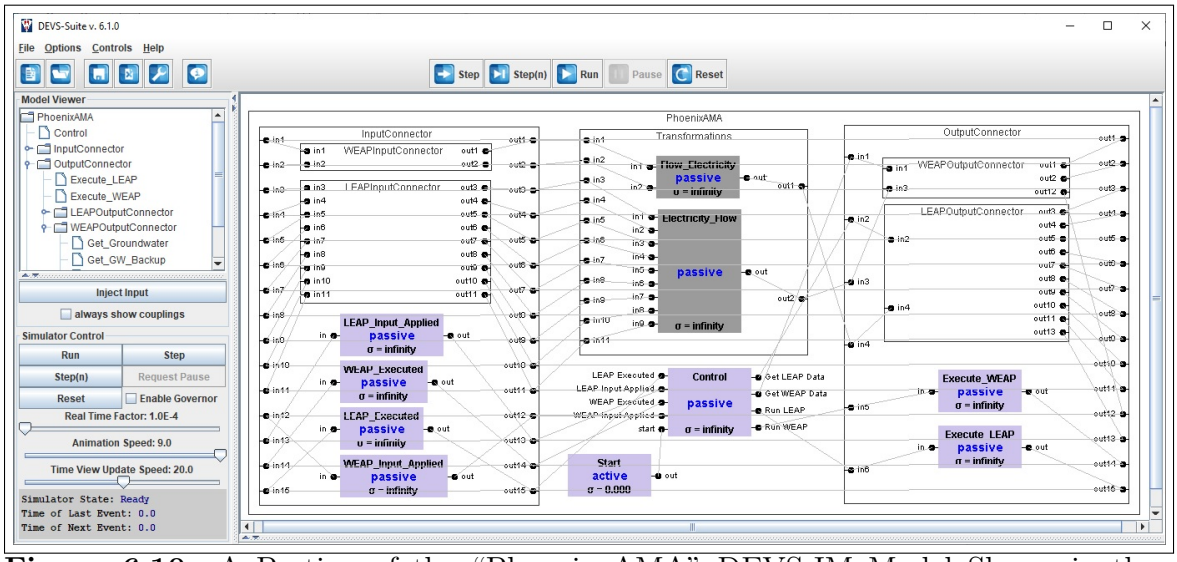


Figure 6.10: A Portion of the "Phoenix AMA" DEVS-IM Model Shown in the DEVS-Suite Simulator.

Figure 6.11 presents a state machine for the "Control" task element of the Phoenix AMA DEVS-IM model. Initially, two `WEAPAppliedCount` and `LEAPAppliedCount` variables (indicate the number of write data on the WEAP and LEAP systems) are set to zero, and the state is changed to *Idle*. By receiving a message on the "start" input port, a message will be sent on the "runWEAP" output port, and the state will be changed to "Running WEAP". As shown in Figure 6.10, the message will be sent to the "Execute_WEAP" output connector, which calls an API from the Componentized WEAP framework to run the WEAP simulation. After completing the execution, a message will be sent on the "out" output port of the "Execute_WEAP"

output connector. This message will be transferred to the "WEAP-Executed" input connector and then will be transferred to the "WEAP Executed" input port of the "Control" model.

As shown in Figure 6.11, the arrived message changes the state from the *Running WEAP* to the *Getting WEAP Data*, and a message is sent on the "Get WEAP Data" output port. The output message will be sent to the output connectors to get data from the WEAP system (by calling the corresponding APIs from the Componentized WEAP framework). The received data from the WEAP system will be sent to the input connectors and then to a task element to apply data transformation (e.g., the "Flow_Electricity" in Figure 6.10). After that, the transformed data will be sent to an output connector to send it to the LEAP system via calling a proper API from the Componentized LEAP framework. Then, the output connector sends an acknowledgment of applying data which will receive on the "LEAP Input Applied" input port of the "Control" model. The `LEAPAppliedCount` variable in Figure 6.11 increments one unit by receiving this message and checks the variable's value. The state will not change if the variable's value is less than the total number of write processes that must be applied to the LEAP system (the value 88 in Figure6.11). Otherwise, the state will be changed to *Running LEAP* and the `LEAPAppliedCount` variable sets to zero. The same scenario happens for running the LEAP system, getting data from the LEAP system, and applying the transformed data to the WEAP system (see Figure 6.10 and Figure 6.11).

6.4 Performance Evaluation of the WEN Modeling Approaches

In this section, the "Phoenix AMA" model is simulated

Table 6.1, Table 6.2, and Figure 6.12 present the allocated time and their order for different execution steps for one round of simulating the "Phoenix AMA" model

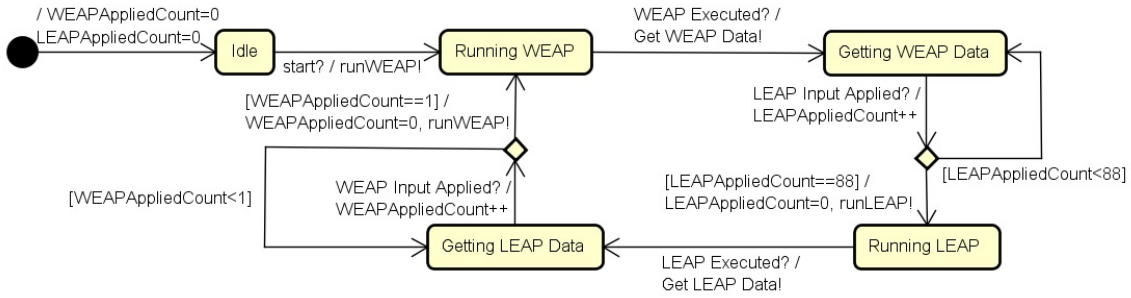


Figure 6.11: A State Machine for the "Control" Task Element of the Phoenix AMA DEVS-IM Model.

using three simulation approaches. In the WEAP-LEAP internal linkage, the WEAP and LEAP systems are running alternatively (the WEAP system runs first in this experiment). As shown in Table 6.1, the execution time of each system includes reading the required data (interconnection between the systems) from the other system (the amount is not distinguishable from the outside), applying the data transformations, then computing the results. As shown in Figure 6.12, the total time for an execution cycle for the WEAP-LEAP internal linkage is $\delta t_w + \delta t_l$. The periods for the water and energy models to read the data from one another are δt_{wr} and δt_{lr} ($0 < \delta t_{wr} < \delta t_w$ and $0 < \delta t_{lr} < \delta t_l$). For one complete simulation round, the WEAP system first reads the LEAP data; then, the water model executes. Next, the LEAP system first reads the WEAP data, and then the energy model executes (see Figure 6.12). The execution time of the Phoenix AMA model using the WEAP-LEAP internal linkage is 394.5 seconds, with around 20% and 80% of computation time consumed by the WEAP and LEAP systems, respectively.

The numbers in Table 6.1 and Table 6.2 are the average of 10 different runs presented in the second unit. The last three columns show the center (median) and the spread of the data (**Quartile 1** and **Quartile 3**). The same experiment has been performed for the same "Phoenix AMA" model (for the internal linkage and

Table 6.1: Time Allocation of Executing the "Phoenix AMA" Model Using WEAP-LEAP Internal Linkage.

	WEAP Execution		LEAP Execution		Total	Quartile 1	Quartile 3
	Read Data from WEAP	Apply Data Transformation	Read Data from LEAP	Apply Data Transformation			
WEAP-LEAP Internal Linkage	76.1		318.4		394.5	389.4	395.7

Algorithmic-IM) using older versions of the WEAP and LEAP system (Fard et al., 2020), which results in different execution times.

As mentioned before, the defined Phoenix AMA models in Algorithmic-IM and DEVS-IM frameworks replicate the execution regime in the WEAP-LEAP internal linkage (ACIMS, 2022d). Table 6.2 presents the order of different steps and their allocated time for the "Phoenix AMA" model simulated via Algorithmic-IM and DEVS-IM approaches. As shown in Figure 6.12, a complete simulation round in both approaches starts by running the WEAP system. Then, the WEAP results are read by calling the proper Componentized WEAP APIs in the DEVS-IM model. In the third step, the data transformations are applied to the received data. Finally, the results are sent to the LEAP system in the fourth step (via calling the Componentized LEAP APIs). The same scenario applies in the other direction, which means running the LEAP system, reading the data from the LEAP system, applying the data transformation, and writing the results to the WEAP system. In both approaches, applying the data transformation in the interaction model takes a negligible amount of time. The significant computation time is for writing data to the LEAP system (84% in both approaches). It was observed in some of our experiments that based on the version of the WEAP, LEAP, and third-party dependencies and the free resources of the hardware/software, this step was as fast as around 600 seconds. Based on our last experiments, the total execution time for one round of the Phoenix AMA model

via the Algorithmic-IM and DEVS-IM are 975.2 and 960.8 seconds, respectively. The Algorithmic-IM and DEVS-IM approaches have around 150% computational overhead compared to the WEAP-LEAP internal linkage approach.

Table 6.2: Time Allocation of Executing the "Phoenix AMA" Model Using Interaction Model.

	WEAP Execution	Read Data from WEAP	IM Data Transformation	Write Data to LEAP	LEAP Execution	Read Data from LEAP	IM Data Transformation	Write Data to WEAP	Total	Quartile 1	Quartile 3
Algorithmic-IM	50	9.6	0.044	824	65.4	24.8	0.025	1.3	975.2	889.9	928.5
DEVS-IM	49.1	9.3	0.044	811.6	65.7	23.7	0.008	1.3	960.8	856.5	896.6

Figure 6.12: Phoenix AMA model execution time allocation via three simulation approaches.

A standalone desktop computer with Windows-10 64-bit OS with four Core i5 Intel CPUs and 20 GB RAM is used for all the experimental results included in this paper. The Componentized WEAP and Componentized LEAP frameworks are implemented using the NodeJS and TypeScript frameworks. The main third-party dependencies used in the frameworks are *Typescript-Node 8.10.2* for using Typescript in the NodeJS server-side application; *Express 4.17.13* is used to build a web application and APIs; *Routing-Controller 0.9.0* is used to create structured, declarative, and organized class-based controllers; *Body-Parser 1.19.1* to parse the incoming request to the web-server,

and *Winax 3.3.4* is used to define ActiveXObject in NodeJS (create WEAP and LEAP instance in server-side applications). The Algorithmic-IM and DEVS-IM frameworks are implemented using Java 11. The Jersey framework is used to call the RESTful web services for the Componentized WEAP and Componentized LEAP frameworks (Kalin, 2013). The WEAP tool (version 2021.0101) and the LEAP tool (version 2020.1.0.56 32-Bit) are used for the above demonstration example.

6.5 Phoenix AMA Model Verification & Validation

As mentioned in Section 3.6, verification ensures that a simulation model is implemented correctly and validation ensures that the model is an accurate representation of the real-world system being simulated. The Phoenix AMA model in the WEAP-LEAP Internal Linkage is validated using *Sensitivity Analysis*, *Expert Review*, and *Cross-Validation*. Applying the *cross-Validation* for the water model of the Phoenix AMA, the historical data for the period 1985–1997 are used to calibrate and the data for the period 1998-2009 are used to validate the model correction and performance. Also, multiple future scenarios are defined for the period 2010-2069 (Guan et al., 2020). Applying the *cross-Validation* for the energy model of the Phoenix AMA, the historical data for the period 2001–2018 are used for model calibration. Multiple future scenarios are defined for the period 2019-2060 (Mounir et al., 2019). Also, both water and energy models for Phoenix AMA are validated by experts from main resource suppliers in Arizona (e.g., SRP and CAP). Modeling the interactions between water and energy systems for Phoenix AMA, the WEAP and LEAP models are used (period 2008-2018 for model calibration, and 2019-2060 for future scenarios) under different spatiotemporal resolutions and coupling configurations (Mounir et al., 2021). In our research, the WEAP-LEAP Internal Linkage model is used as the base model for simulation validation of the interaction models. In other words, the

validated WEAP-LEAP Internal Linkage model is used for validating the interaction models (i.e., the Algorithmic-IM and DEVS-IM frameworks).

The *Code Review*, *Analytical V&V*, and *Experimentation* methods are used in this research to verify and validate the defined models and their simulation in the presented interaction models (see Section 3.6). The generated code in the code generation phase of the DEVS-IM framework (see Section 5.3.5) is verified precisely. Also, the DEVS-Suite Simulator has a robust simulator, so the executable generated codes for the DEVS-Suite (via DEVS-IM framework) verify individual defined models/components. Figure 6.13a shows the mathematical schema in the WEAP-LEAP Internal Linkage. As mentioned in Section 2.4.1 and Section 2.4.2, a model in the WEAP/LEAP is defined via a set of specific components (Entity A, Entity B, and Entity C in Figure 6.13a), and each component has a set of variables (WV and LV stand for the WEAP Variable and LEAP Variable in Figure 6.13a). Each formula must be implemented inside the WEAP or LEAP components, and it can be a function of different variables in the WEAP and/or LEAP systems. For example, WV_i in Figure 6.13a is defined as $c_1 \times LV_p + C_2 \times WV_o + c_3$ which means it is reading the LV_p from the LEAP system and WV_o from the WEAP system and using some constants. There is a $Read(Variable_{target})$ function between the WEAP and LEAP systems representing the reading data from the other system. For example, LV_k is reading WV_n and WV_m from the WEAP system (see the defined formula for LK_v in Entity C). In this approach, each system has the control to execute the simulation and read the data from the other system (see the gear icons in the WEAP and LEAP systems in Figure 6.13a).

An implementation of the formulas in Figure 6.13a is presented in Figure 6.13b which transfer the whole WV_i and LV_k and part of the WV_j formulas to the Interaction Model. Variables are defined in the Interaction Model (i.e., IM_x, IM_y, and IM_z) to

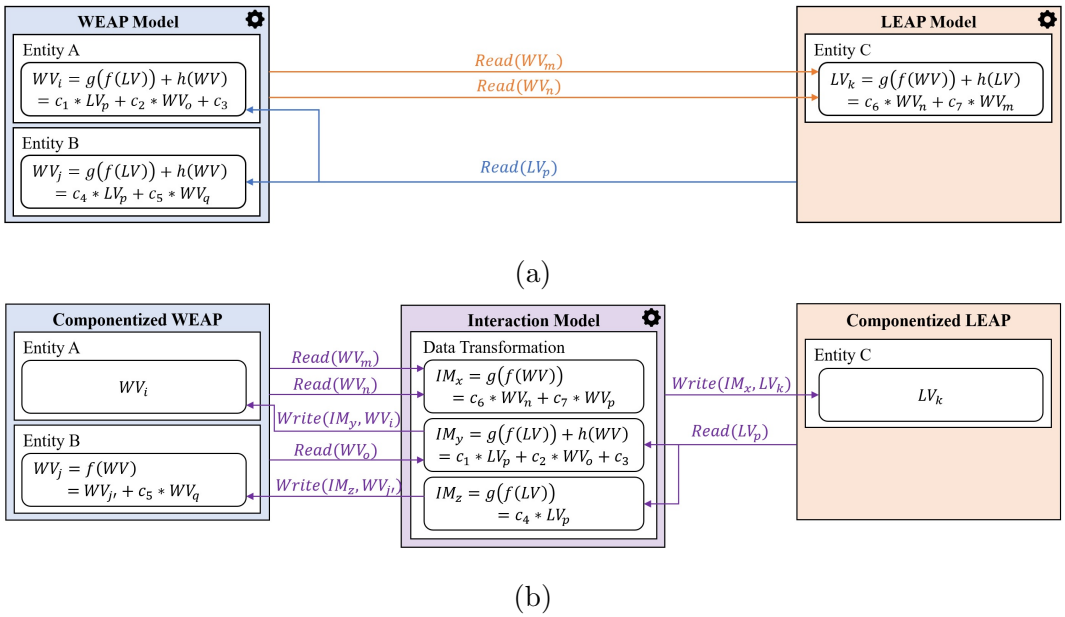


Figure 6.13: The mathematical schema for a defined model in the (a) WEAP-LEA P Internal Linkage and (b) Interaction Model (i.e., Algorithmic-IM and DEVS-IM).

hold the results of the computation. In this approach, the WEAP and LEAP systems are slaves and the Interaction Model is the master to control the simulation execution (see the gear icon in the Interaction Model in Figure 6.13b). The WEAP and LEAP systems do not have any capability to read the data from the other system. There are two functions controlled by the Interaction Model (i.e., $Read(Variable_{target})$ and $Write(Variable_{source}, Variable_{target})$) representing reading/writing data from/to the components of the connected WEAP and LEAP systems.

The validated WEAP-LEAP Internal Linkage model is used for validating the interaction models. Figure 6.14 shows a simple procedure where data sets are collected from the Internal Linkage and DEVS-IM. In the first step, the defined model using the WEAP-LEAP Internal Linkage and DEVS-IM. The DEVS-IM model is using the same water and energy models defined in the WEAP-LEAP Internal Linkage, except removing the direct internal connection between the models. In the second step, a specific set of outputs from both simulations are collected in CSV files (using JScript

and Java for WEAP-LEAP Internal Linkage and DEVS-IM, respectively). A CSV file contains the simulation results (i.e., the values for different years and timesteps of the simulation) for a specific project, component, scenario, and variable. In the third step, a validator (written in Java) compares the corresponding values.

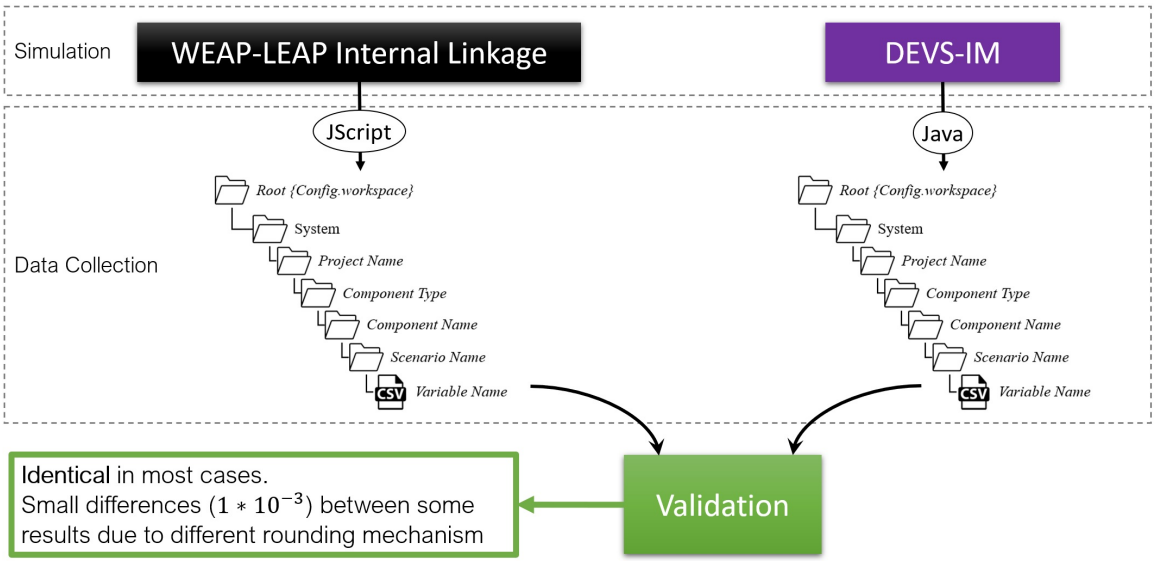


Figure 6.14: DEVS-IM model simulation validation using corresponding WEAP-LEAP Internal Linkage model.

The output of the validation for the Phoenix AMA model shows that the results of the simulation executions are identical in most cases (for the values which have less than 15 digits in the WEAP and LEAP models). There are some small differences (around 1×10^{-3}) between some results (for the values which have more than 15 digits in the WEAP and LEAP models). The negligible differences are because of the rounding mechanism in using the WEAP and LEAP APIs. The internal linkage of the WEAP and LEAP systems transfers data with complete precision, whereas the WEAP and LEAP APIs export data with a maximum of 15-digit precision. So, the task components (to transform the data) in the DEVS-IM model receive data with a maximum of 15-digit precision.

Chapter 7

CONCLUSION & FUTURE WORK

7.1 Conclusion

The WEAP/LEAP system is appealing to domain experts from the standpoint of ease of use for rapid model development. This research provides detail for defining the WEAP/LEAP components as proxies for the WEAP/LEAP entities using meta-modeling and Model Driven Architecture. The WEAP/LEAP entities, input and output variables, and their data are represented using the Ecore meta-modeling approach, where each proxy model component corresponds to a WEAP/LEAP entity. These components are useful for service-oriented modeling and simulation frameworks. The outcome is the Componentized WEAP/LEAP RESTful framework which helps to consider a set of component models instead of thinking about a group of shared variables (belonging to different entities) that are used in mass-balanced equations. Also, the REST APIs ease the use of the WEAP/LEAP system in modern computing platforms, including its integration with other tools to model and simulate complex systems such as the Phoenix Active Management Area. The componentization of the WEAP/LEAP systems supports a higher degree of control for manipulating and simulating the water and energy entity models. For example, the Componentized WEAP/LEAP framework can help simplify the design of simulation experiments and optimization studies that can be difficult using the scripting languages supported in the WEAP system. The realization of the Componentized WEAP/LEAP RESTful framework can be adopted to simplify their integration with other web-services.

The Phoenix Active Management Area (Phoenix AMA) serves as the case study in this research. A model was developed in the WEAP tool to define the water and food systems of the Phoenix AMA. Another model was developed in the LEAP system to define the energy system of the Phoenix AMA. These models are connected via WEAP-LEAP internal linkage, Algorithmic-IM framework, and DEVS-IM framework. The simulation results of the Algorithmic-IM and DEVS-IM frameworks are validated using the WEAP-LEAP internal linkage. The computational cost of simulating the Phoenix AMA model using the interaction models (Algorithmic-IM and DEVS-IM) is about twice the cost of the data sharing allowed in the WEAP-LEAP internal linkage. However, the benefit of composing hybrid water and energy models with interaction models outweighs its higher computational cost as compared with data sharing, especially when the key consideration is flexible model composability, not the amount of time it takes to simulate hybrid models.

7.2 Future Work

The Componentized WEAP and Componentized LEAP frameworks support reading existing models from the WEAP and LEAP systems, respectively. Enabling these frameworks to create/update models via APIs can allow using the WEAP and LEAP systems remotely and in distributed settings. The Algorithmic-IM and DEVS-IM models can be created using textual programming language and REST APIs. Instead, visual modeling of FEW-nexus is attractive for domain experts or those who prefer not to write code. The Componentized WEAP, Componentized LEAP, and Algorithmic-IM/DEVS-IM with the proposed unified persistence and visual modeling capability are important for developing and using Food-Energy-Water systems. Visual modeling frameworks for Parallel DEVS exist and can be utilized (ACIMS, 2022a; Sarjoughian & Elamvazhuthi, 2010; Sarjoughian et al., 2011).

During this research, the food system was embedded inside the water model (using the WEAP system). In the Phoenix-AMA model, the food model is used as input. It is beneficial to have a separate model for the food system. This can allow bi-directional interactions between the water and the energy model. The Algorithmic-IM and DEVS-IM frameworks have an abstract specification to define an interaction model for disparate systems that form systems-of-systems. From the structure specification point of view, adding the Food system to the current interaction model should be straightforward. However, it is necessary to verify and validate the execution of the whole Food-Energy-Water Nexus for an actual system such as the Phoenix AMA.

The set of Logic elements for the DEVS-IM must be designed and implemented. The DEVS-IM framework has some considerations for the Logic elements (e.g., the Choice, Junction, and Sync). A complete predefined set of logic elements in the DEVS-IM framework can offer more support for model development.

REFERENCES

ACIMS. (2022a). Component-based System Modeling and Simulator. *Arizona Center for Integrative Modeling & Simulation*. https://acims.asu.edu/modsim/cosmos/ (accessed 04-11-2023)

ACIMS. (2022b). *Componentized WEAP RESTful Framework*. Arizona Center for Integrative Modeling & Simulation. https://acims.asu.edu/modsim/weap-kib-leap (accessed: 03.20.2023)

ACIMS. (2022c). DEVS-Suite Simulator. *Arizona Center for Integrative Modeling & Simulation*. https://acims.asu.edu/devs-suite/ (accessed 04-11-2022)

ACIMS. (2022d). DEVS-Suite Simulator. *Arizona Center for Integrative Modeling & Simulation*. https://acims.asu.edu/devs-suite/ (accessed 04-11-2022)

Agrawal, N., Ahiduzzaman, M., & Kumar, A. (2018). The development of an integrated model for the assessment of water and GHG footprints for the power generation sector. *Applied Energy*, *216*, 558–575.

Akhbari, M. (2012). *Models for management of water conflicts: A case study of the San-Joaquin watershed, California* (Doctoral dissertation). Colorado State University.

Alvi, M. S. Q., Mahmood, I., Javed, F., Malik, A. W., & Sarjoughian, H. (2018). Dynamic behavioral modeling, simulation, and analysis of household water consumption in an urban area: a hybrid approach. *2018 Winter Simulation Conference (WSC)*, 2411–2422.

Amin, A., Iqbal, J., Asghar, A., & Ribbe, L. (2018). Analysis of current and future water demands in the Upper Indus Basin under IPCC climate and socio-economic scenarios using a hydro-economic WEAP model. *Water*, *10*(5), 537.

Baki, S., & Makropoulos, C. (2014). Tools for energy footprint assessment in urban water systems. *Procedia Engineering*, *89*, 548–556.

Banks, J. (1998). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. John Wiley & Sons.

Barton, C. M., Ullah, I. I., & Bergin, S. (2010). Land use, water, and Mediterranean landscapes: modeling long-term dynamics of complex socio-ecological systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *368*(1931), 5275–5297.

Barton, C. M., Ullah, I. I., Bergin, S. M., Sarjoughian, H. S., Mayer, G. R., Bernabeu-Auban, J. E., Heimsath, A. M., Acevedo, M. F., Riel-Salvatore, J. G., & Arrowsmith, J. R. (2016). Experimental Socioecology: Integrative Science for Anthropocene Landscape Dynamics. *Anthropocene*, *13*, 34–45.

Bazilian, M., Rogner, H., Howells, M., Hermann, S., Arent, D., Gielen, D., Steduto, P., Mueller, A., Komor, P., Tol, R. S., et al. (2011). Considering the energy, water, and food nexus: Towards an integrated modeling approach. *Energy policy*, *39*(12), 7896–7906.

Bhatnagar, S., & Narang, R. (2014). Model Verification and Validation in Simulation. *International Journal of Engineering Research & Technology*, *3*(9), 852–856.

Biggs, E. M., Bruce, E., Boruff, B., Duncan, J. M., Horsley, J., Pauli, N., McNeill, K., Neef, A., Van Ogtrop, F., Curnow, J., et al. (2015). Sustainable development and the water-energy-food nexus: A perspective on livelihoods. *Environmental Science & Policy*, *54*, 389–397.

Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999). Legacy information systems: Issues and directions. *IEEE software*, *16*(5), 103–111.

BMI-2 Documentation. (2022). https://bmi.readthedocs.io/en/stable/ (accessed: 11.20.2022)

Boretti, A., & Rosa, L. (2019). Reassessing the projections of the world water development report. *NPJ Clean Water*, *2*(1), 1–6.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., & Winer, D. (2000). Simple Object Access Protocol (SOAP) 1.1.

Boyd, W. A., & Sarjoughian, H. S. (2020). Composition of Geographic-Based Component Simulation Models. *2020 Winter Simulation Conference (WSC)*, 2257–2268.

Brutzman, D., Zyda, M., Pullen, J. M., & Morse, K. L. (2002). Extensible Modeling and Simulation Framework (XMSF): Challenges for Web-based modeling and simulation. *Findings and Recommendations Report of the XMSF Technical Challenges Workshop and Strategic Opportunities Symposium*.

Budinsky, F., Ellersick, R., Steinberg, D., Grose, T. J., & Merks, E. (2004). *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley Professional.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007). *Pattern-oriented software architecture, on patterns and pattern languages*. John wiley & sons.

Cantelon, M., Harter, M., Holowaychuk, T., & Rajlich, N. (2014). *Node.js in Action*. Manning Greenwich.

Castronova, A. M., Goodall, J. L., & Elag, M. M. (2013). Models as web services using the Open Geospatial Consortium (OGC) Web Processing Service (WPS) standard. *Environmental Modelling & Software*, *41*, 72–83.

Chow, A. C. H., & Zeigler, B. P. (1994). Parallel DEVS: A parallel, hierarchical, modular modeling formalism. *Proceedings of Winter Simulation Conference*, 716–722.

CLEWS-Home. (2022). http://www.osimosys.org (accessed: 09.20.2022)

Cozzi, L., Gould, T., Bouckart, S., Crow, D., Kim, T., McGlade, C., Olejarnik, P., Wanner, B., & Wetzel, D. (2020). World Energy Outlook 2020. *vol*, *2050*, 1–461.

Daher, B. T., & Mohtar, R. H. (2015). Water-energy-food (WEF) Nexus Tool 2.0: guiding integrative resource planning and decision-making. *Water International*, *40*(5-6), 748–771.

Dahmann, J. S., Fujimoto, R. M., & Weatherly, R. M. (1997). The Department of Defense High Level Architecture. *Proceedings of the 29th conference on Winter simulation*, 142–149.

Dai, J., Wu, S., Han, G., Weinberg, J., Xie, X., Wu, X., Song, X., Jia, B., Xue, W., & Yang, Q. (2018). Water-energy nexus: A review of methods and tools for macro-assessment. *Applied energy*, *210*, 393–408.

Dale, A. T., & Bilec, M. M. (2014). The Regional Energy & Water Supply Scenarios (REWSS) model, part I: framework, procedure, and validation. *Sustainable Energy Technologies and Assessments*, *7*, 227–236.

Dale, L. L., Karali, N., Millstein, D., Carnall, M., Vicuña, S., Borchers, N., Bustos, E., O'hagan, J., Purkey, D., Heaps, C., et al. (2015). An integrated assessment of water-energy and climate change in Sacramento, California: how strong is the nexus? *Climatic Change*, *132*(2), 223–235.

Darbandsari, P., Kerachian, R., & Malakpour-Estalaki, S. (2017). An Agent-based Behavioral Simulation Model for Residential Water Demand Management: The Case-Study of Tehran, Iran. *Simulation Modelling Practice and Theory*, *78*, 51–72.

Dargin, J., Daher, B., & Mohtar, R. H. (2019). Complexity versus simplicity in water energy food nexus (WEF) assessment tools. *Science of the Total Environment*, *650*, 1566–1575.

Davis, P. K., & Anderson, R. H. (2004). Improving the composability of DoD models and simulations. *The Journal of Defense Modeling and Simulation*, *1*(1), 5–17.

Fard, M. D., & Sarjoughian, H. (2019). A Web-Service Framework for the Water Evaluation and Planning System. *2019 Spring Simulation Conference (SpringSim)*, 1–12.

Fard, M. D., & Sarjoughian, H. S. (2020). Coupling WEAP and LEAP models using interaction modeling. *2020 Spring Simulation Conference (SpringSim)*, 1–12.

Fard, M. D., & Sarjoughian, H. S. (2021a). A RESTful framework design for componentizing the Water evaluation and planning (WEAP) system. *Simulation Modelling Practice and Theory*, *106*, 102199.

Fard, M. D., & Sarjoughian, H. S. (2021b). A RESTful persistent DEVS-based interaction model for the Componentized WEAP and LEAP RESTful frameworks. *2021 Winter Simulation Conference (WSC)*, 1–12.

Fard, M. D., & Sarjoughian, H. S. (2022). Model And Simulation Scalability Traits for Interaction (Nexus) Modeling Of Water And Energy Systems. *2022 Annual Modeling and Simulation Conference (ANNSIM)*, 437–448.

Fard, M. D., Sarjoughian, H. S., Mahmood, I., Mounir, A., Guan, X., & Mascaro, G. (2020). Modeling the Water-Energy Nexus for the Phoenix Active Management Area. *2020 Winter Simulation Conference (WSC)*, 2317–2328.

Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, *2*(2), 115–150.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

Gao, F., Yue, P., Zhang, C., & Wang, M. (2019). Coupling components and services for integrated environmental modelling. *Environmental modelling & software*, *118*, 14–22.

Gao, J., Christensen, P., & Li, W. (2017). Application of the WEAP model in strategic environmental assessment: Experiences from a case study in an arid/semi-arid area in China. *Journal of environmental management*, *198*, 363–371.

Geissdoerfer, M., Savaget, P., Bocken, N. M., & Hultink, E. J. (2017). The Circular Economy-A new sustainability paradigm? *Journal of cleaner production*, *143*, 757–768.

Giampietro, M., Aspinall, R. J., Bukkens, S. G., Benalcazar, J. J. C., Maurin, F. D., Flammini, A., Gomiero, T., Kovacic, Z., Madrid-Lopez, C., Martin, J. R., et al. (2013). An Innovative Accounting Framework for the Food-Energy-Water Nexus: Application of the MuSIASEM Approach to Three Case Studies.

Gondhalekar, D., & Ramsauer, T. (2017). Nexus City: Operationalizing the urban Water-Energy-Food Nexus for climate change adaptation in Munich, Germany. *Urban Climate*, *19*, 28–40.

Goodall, J. L., Robinson, B. F., & Castronova, A. M. (2011). Modeling water resource systems using a service-oriented computing paradigm. *Environmental Modelling & Software*, *26*(5), 573–582.

Guan, X., Mascaro, G., Sampson, D., & Maciejewski, R. (2020). A metropolitan scale water management analysis of the food-energy-water nexus. *Science of The Total Environment*, *701*, 134478.

Gurobi. (2022). *Gurobi Optimization*. https://www.gurobi.com (accessed: 09.20.2022)

Harpham, Q. K., Hughes, A., & Moore, R. (2019). Introductory overview: The OpenMI 2.0 standard for integrating numerical models. *Environmental Modelling & Software*, *122*, 104549.

Helton, J. C., & Davis, F. J. (2003). Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliability Engineering & System Safety*, *81*(1), 23–69.

Helton, J. C. (1993). Uncertainty and sensitivity analysis techniques for use in performance assessment for radioactive waste disposal. *Reliability Engineering & System Safety*, *42*(2-3), 327–367.

Hoff, H. (2011). Understanding the Nexus. Background Paper for the Bonn2011 Nexus Conference.

Höllermann, B., Giertz, S., & Diekkrüger, B. (2010). Benin 2025—Balancing future water availability and demand using the WEAP 'Water Evaluation and Planning'System. *Water resources management*, *24*(13), 3591–3613.

Howells, M., Hermann, S., Welsch, M., Bazilian, M., Segerström, R., Alfstad, T., Gielen, D., Rogner, H., Fischer, G., Van Velthuizen, H., et al. (2013). Integrated analysis of climate change, land-use, energy, and water strategies. *Nature Climate Change*, *3*(7), 621–626.

Huang, D. (2008). *Composable modeling and distributed simulation framework for discrete supply-chain systems with predictive control* (Doctoral dissertation). Arizona State University.

Huang, D., Sarjoughian, H. S., Wang, W., Godding, G., Rivera, D. E., Kempf, K. G., & Mittelmann, H. (2009). Simulation of semiconductor manufacturing supply-chain systems with DEVS, MPC, and KIB. *IEEE Transactions on Semiconductor Manufacturing*, *22*(1), 164–174.

Hutton, E. W., Piper, M. D., & Tucker, G. E. (2020). The Basic Model Interface 2.0: A standard interface for coupling numerical models in the geosciences. *Journal of Open Source Software*, *5*(51), 2317.

Initiative, W. E. F. W., et al. (2012). *Water Security: The Water-Food-Energy-Climate Nexus*. Island Press.

Institute, S. E. (2022). *Introduction to Optimization*. https://leap.sei.org/help/Optimization/OptimizationIntroduction.htm (accessed: 09.20.2022)

Islam, S., & Karim, Z. (2019). World's demand for food and water: The consequences of climate change. *Desalination-challenges and opportunities*, 57–84.

Javadifard, N., Khadivi, S., Motahari, S., & Farahani, M. (2020). Modeling of water-energy-environment nexus by water evaluation and planning and long-range energy alternative planning models: A case study. *Environmental Progress & Sustainable Energy*, *39*(2), e13323.

Jiang, P., Elag, M., Kumar, P., Peckham, S. D., Marini, L., & Rui, L. (2017). A service-oriented architecture for coupling web service models using the Basic Model Interface (BMI). *Environmental Modelling & Software*, *92*, 107–118.

Kaddoura, S., & El Khatib, S. (2017). Review of water-energy-food Nexus tools to improve the Nexus modeling approach for integrated policy making. *Environmental Science & Policy*, *77*, 114–121.

Kalin, M. (2013). *Java Web Services: Up and Running: A Quick, Practical, and Thorough Introduction.* " O'Reilly Media, Inc.".

Keairns, D., Darton, R., & Irabien, A. (2016). The Energy-Water-Food Nexus. *Annual review of chemical and biomolecular engineering*, *7*, 239–262.

Kenney, J. L. (2017). Model Verification and Validation. In *The engineering handbook* (2nd ed.). CRC Press.

Khosrowjerdi, M. (2020). Model Verification and Validation in Simulation. *Journal of Physics: Conference Series*, *1529*(1). https://doi.org/10.1088/1742-6596/1529/1/012030

Kim, S., Sarjoughian, H. S., & Elamvazhuthi, V. (2009). DEVS-Suite: A simulator supporting visual experimentation design and behavior monitoring. *Proceedings of the 2009 Spring Simulation Multiconference*, 1–7.

Kohavi, R. (1995). A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. *International Joint Conference on Artificial Intelligence*.

Law, A. M., & Kelton, W. D. (2000). *Simulation Modeling and Analysis* (Vol. 1). McGraw-Hill.

Law, A. M. (2019). How to build valid and credible simulation models. *2019 Winter Simulation Conference (WSC)*, 1402–1414.

Lazarus, M., Greber, L., Hall, J., Bartels, C., Bernow, S., Hansen, E., Raskin, P., & Von Hippel, D. (1993). Towards a fossil-free energy future. The next energy transition.

Lévite, H., Sally, H., & Cour, J. (2003). Testing water demand management scenarios in a water-stressed basin in South Africa: application of the WEAP model. *Physics and Chemistry of the Earth, Parts A/B/C*, *28*(20-27), 779–786.

Li, W., Wu, S., Song, M., & Zhou, X. (2016). A scalable cyberinfrastructure solution to support big data management and multivariate visualization of time-series sensor observation data. *Earth Science Informatics*, *9*(4), 449–464.

Ma, X., Yang, D., Shen, X., Zhai, Y., Zhang, R., & Hong, J. (2018). How much water is required for coal power generation: An analysis of gray and blue water footprints. *Science of the Total Environment*, *636*, 547–557.

Markstrom, S. L., Niswonger, R. G., Regan, R. S., Prudic, D. E., & Barlow, P. M. (2008). GSFLOW-Coupled Ground-water and Surface-water FLOW model based on the integration of the Precipitation-Runoff Modeling System (PRMS) and the Modular Ground-Water Flow Model (MODFLOW-2005). *US Geological Survey techniques and methods*, *6*, 240.

Markstrom, S. L., Regan, R. S., Hay, L. E., Viger, R. J., Webb, R. M., Payn, R. A., & LaFontaine, J. H. (2015). PRMS-IV, the precipitation-runoff modeling system, version 4. *US Geological Survey Techniques and Methods*, *6*, B7.

Martinez-Hernandez, E., Leach, M., & Yang, A. (2017). Understanding water-energy-food and ecosystem interactions using the nexus simulation tool NexSym. *Applied Energy*, *206*, 1009–1021.

Mayer, G. R. (2009). *Composing hybrid discrete event system and cellular automata models* (Doctoral dissertation). Arizona State University.

Mayer, G. R., & Sarjoughian, H. S. (2007). Complexities of simulating a hybrid agent-landscape model using multi-formalism composability. *Agent-Directed Simulation, Spring Simulation Multiconference, Norfolk, Virginia, pp. 161-168, March*.

McLaughlin, M. B., & Sarjoughian, H. S. (2020). DEVS-scripting: A Black-Box Test Frame for DEVS Models. *2020 Winter Simulation Conference (WSC)*, 2196–2207.

Mounir, A., Guan, X., & Mascaro, G. (2021). Investigating the value of spatiotemporal resolutions and feedback loops in water-energy nexus modeling. *Environmental Modelling & Software*, *145*, 105197.

Mounir, A., Mascaro, G., & White, D. D. (2019). A metropolitan scale analysis of the impacts of future electricity mix alternatives on the water-energy nexus. *Applied Energy*, *256*, 113870.

MPI. (2022). *MPI Forum*. https://www.mpi-forum.org (accessed: 09.20.2022)

Nanduri, V., & Saavedra-Antolnez, I. (2013). A competitive Markov decision process model for the energy-water-climate change nexus. *Applied energy*, *111*, 186–198.

Ni, J., Liu, M., Ren, L., & Yang, S. X. (2013). A Multiagent Q-Learning-Based Optimal Allocation Approach for Urban Water Resource Management System. *IEEE Transactions on Automation Science and Engineering*, *11*(1), 204–214.

Nikolic, V. V., & Simonovic, S. P. (2015). Multi-method modeling framework for support of integrated water resources management. *Environmental Processes*, *2*(3), 461–483.

Oberkampf, W. L., & Trucano, T. G. (2002). Verification and validation in computational fluid dynamics. *Progress in aerospace sciences*, *38*(3), 209–272.

Parr, T. (2022). *StringTemplate*. https://www.stringtemplate.org (accessed: 09.20.2022)

Pautasso, C., & Wilde, E. (2010). RESTful web services: principles, patterns, emerging technologies. *Proceedings of the 19th international conference on World wide web*, 1359–1360.

Petty, M. D., Kim, J., Barbosa, S. E., & Pyun, J.-J. (2014). Software frameworks for model composition. *Modelling and Simulation in Engineering*, *2014*.

Proctor, K., Tabatabaie, S. M., & Murthy, G. S. (2021). Gateway to the perspectives of the Food-Energy-Water nexus. *Science of The Total Environment*, *764*, 142852.

Psomas, A., Panagopoulos, Y., Konsta, D., & Mimikou, M. (2016). Designing water efficiency measures in a catchment in Greece using WEAP and SWAT models. *Procedia engineering*, *162*, 269–276.

Pullen, J. M., Brunton, R., Brutzman, D., Drake, D., Hieb, M., Morse, K. L., & Tolk, A. (2005). Using Web Services to Integrate Heterogeneous Simulations in a Grid Environment. *Future Generation Computer Systems*, *21*(1), 97–106.

Qin, F., Zhu, J., Wang, H., et al. (2019). An object-oriented framework for modeling watershed flow and sediment process based on fine-grained components. *Arabian Journal of Geosciences*, *12*(19), 1–14.

Richardson, L., & Ruby, S. (2008). *RESTful web services*. " O'Reilly Media, Inc.".

Roache, P. J. (1998). *Verification and validation in computational science and engineering* (Vol. 895). Hermosa Albuquerque, NM.

Roy, P., Nei, D., Orikasa, T., Xu, Q., Okadome, H., Nakamura, N., & Shiina, T. (2009). A review of life cycle assessment (LCA) on some food products. *Journal of food engineering*, *90*(1), 1–10.

Saladini, F., Betti, G., Ferragina, E., Bouraoui, F., Cupertino, S., Canitano, G., Gigliotti, M., Autino, A., Pulselli, F., Riccaboni, A., et al. (2018). Linking the water-energy-food nexus and sustainable development indicators for the Mediterranean region. *Ecological Indicators*, *91*, 689–697.

Sargent, R. G. (2010). Verification and validation of simulation models. *Proceedings of the 2010 winter simulation conference*, 166–183.

Sarjoughian, H. S. (2006). Model Composability. *Proceedings of the 2006 Winter Simulation Conference*, 149–158.

Sarjoughian, H. S., & Elamvazhuthi, V. (2010). CoSMoS: a visual environment for component-based modeling, experimental design, and simulation. *2nd International ICST Conference on Simulation Tools and Techniques*.

Sarjoughian, H. S., Nutaro, J. J., & Joshi, G. (2011). Towards collaborative component-based modeling. *Journal of Simulation*, *5*, 77–88.

Sarjoughian, H. S., & Singh, R. (2004). Building simulation modeling environments using systems theory and software architecture principles. *Proceedings of the Advanced Simulation Technology conference*, 99–104.

Sarjoughian, H. S., Smith, J., Godding, G. W., & Muqsith, M. A. (2013). Model composability and execution across simulation, optimization, and forecast models. *SpringSim (TMS-DEVS)*, 30.

Sarjoughian, H. S., & Zeigler, B. P. (2000). DEVS and HLA: Complementary paradigms for modeling and simulation? *TRANSACTIONS of the Society for Computer Simulation*, *17*(4), 187–197.

SEI. (2022a). *Automating WEAP (API)*. http://www.weap21.org/WebHelp/API.htm (accessed: 09.20.2022)

SEI. (2022b). *Low Emission Analysis Platform (LEAP)*. https://leap.sei.org/ (accessed: 09.20.2022)

SEI. (2022c). *NEMO Documentation*. https://sei-international.github.io/NemoMod.jl/stable (accessed: 09.20.2022)

SEI. (2022d). *WEAP: Water Evaluation And Planning System*. http://www.weap21.org/index.asp (accessed: 09.20.2022)

Serraj, R., & Pingali, P. (2018). Agriculture & Food Systems to 2050: Global Trends, Challenges, and Opportunities.

Siddiqi, A., & Anadon, L. D. (2011). The water-energy nexus in Middle East and North Africa. *Energy policy*, *39*(8), 4529–4540.

Sieber, J., Swartz, C., & Huber-Lee, A. (2005). Water Evaluation and Planning System (WEAP): User Guide. *Stockholm Environment Institute, Boston*.

Smith, J. M. (2012). *Scalable Knowledge Interchange Broker: Design and Implementation for Semiconductor Supply Chain Systems* (Master's thesis). Arizona State University.

Sneed, H. M. (2006). Integrating legacy software into a service-oriented architecture. *Conference on Software Maintenance and Reengineering (CSMR'06)*, 11–pp.

Sneed, H. M., et al. (2006). Wrapping legacy software for reuse in a SOA. *Multikonferenz Wirtschaftsinformatik*, *2*, 345–360.

Sonntag, M., Hotta, S., Karastoyanova, D., Molnar, D., & Schmauder, S. (2011). Using services and service compositions to enable the distributed execution of legacy simulation applications. *European Conference on a Service-Based Internet*, 242–253.

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

Thacker, B. H., Doebling, S. W., Hemez, F. M., Anderson, M. C., Pepin, J. E., & Rodriguez, E. A. (2004). Concepts of model verification and validation.

Tihomirovs, J., & Grabis, J. (2016). Comparison of soap and REST-based web services using software evaluation metrics. *Information technology and management science*, *19*(1), 92–97.

Tukey, J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley.

Using Express Middleware. (2022). https://expressjs.com/en/guide/using-middleware.html (accessed: 09.20.2022)

Wainwright, R. S., & Sivilotti, P. L. P. (2018). Data validation for simulation models: A review and perspective. *Simulation Modelling Practice and Theory*, *87*. https://doi.org/10.1016/j.simpat.2018.07.005

Wittgenstein, L. (2012). *The big typescript: TS 213*. John Wiley & Sons.

Xia, Y., & Yan, B. (2022). Energy-food nexus scarcity risk and the synergic impact of climate policy: A global production network perspective. *Environmental Science & Policy*, *135*, 26–35.

Yates, D., Sieber, J., Purkey, D., & Huber-Lee, A. (2005). WEAP21—A demand-, priority-, and preference-driven water planning model: part 1: model characteristics. *Water international*, *30*(4), 487–500.

Yaung, S. J., Church, G. M., & Wang, H. H. (2014). Recent progress in engineering human-associated microbiomes. *Engineering and Analyzing Multicellular Systems: Methods and Protocols*, 3–25.

Zeigler, B. P., Muzy, A., & Kofman, E. (2018). *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. Academic press.

Zeigler, B. P., Sarjoughian, H. S., Duboz, R., & Soulie, J.-C. (2017). *Guide to modeling and simulation of systems of systems*. Springer.

Zengin, A. (2010). Large-scale integrated network system simulation with DEVS-Suite. *KSII Transactions on Internet and Information Systems (TIIS)*, *4*(4), 452–474.

Zhang, C., Sarjoughian, H. S., & Seok, M. G. (2020). A framework for composable cellular automata DEVS modeling, simulation, and visualization. *2020 Spring Simulation Conference (SpringSim)*, 1–12.

Zhang, P., Zhang, L., Chang, Y., Xu, M., Hao, Y., Liang, S., Liu, G., Yang, Z., & Wang, C. (2019). Food-energy-water (FEW) nexus for urban sustainability: A comprehensive review. *Resources, Conservation and Recycling*, *142*, 215–224.

APPENDIX A

WEAP SCRIPTING APIS

Table A.1: The WEAP's Scripting APIs Used in the Componentized WEAP Framework

Category	Scripting API	Return Object
WEAP	WEAP.ActiveArea	Area
	WEAP.ActiveArea.Name	String
	WEAP.WaterYearStart	Integer
	WEAP.ActiveScenario	Scenario
	WEAP.BaseYear	Integer
	WEAP.EndYear	Integer
	WEAP.TimeStepName(Id)	String
	WEAP.NumTimeSteps	Integer
	WEAP.View	String
	WEAP.Calculate(LastYear, LastTimestep, AlwaysCalculate)	∅
	WEAP.ResultValue(BranchName:VariableName, Year, TimeStep,ScenarioName)	Double
Area	WEAP.Areas(Id)	Area[]
	WEAP.Areas.Count	Integer
Version	WEAP.Versions.Count	Integer
	WEAP.Versions(Name/Id)	Version
	WEAP.Versions.Exist(VersionName)	Boolean
	WEAP.SaveVersion(VersionName)	∅
	WEAP.Versions(VersionName).Revert()	∅
Scenario	WEAP.Scenarios(Id)	Scenario[]
	WEAP.Scenarios.Exists(ScenarioName)	Boolean
	WEAP.Scenarios.Add(ScenarioName)	∅
	WEAP.Scenarios(ScenarioName).Delete()	∅
Branch	WEAP.Branch(BranchName)	Branch
	WEAP.BranchExists(BranchName)	Boolean
	WEAP.Branch(BranchName).Children	Branch[]
	WEAP.Branch(BranchName).Variables	Variable[]
	WEAP.Branch(BranchName).Variables.Exists(VariableName)	Boolean

APPENDIX B

DEVS-IM TEMPLATE FILES TO GENERATE DEVS-SUITE CODES

The code generation module of the DEVS-IM framework to the DEVS-Suite simulator is based on using the StringTemplate library (Parr, 2022), a template engine for generating source code, web pages, emails, or any other formatted text output. Listing B.1 to Listing B.10 present the content of the template files for different type of DEVS-IM elements.

Listing B.1: The Content of the IM.stg Template File of the DEVS-IM Framework.

```
IM(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
    Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import java.awt.*;
import view.modeling.ViewableAtomic;
import view.modeling.ViewableDigraph;
import view.modeling.ViewableComponent;
import InteractionModel.core.Process;
<component.imports:AddImport(); separator = "\n">

/*
 * id: <component.id>
 * componentType: <component.componentType>
 * description: <component.description>
 */
public class <component.name> extends Process {
 public <component.name>() {
  super("<component.name>");
  <InsertBlank(component.connectors)>
  <component.connectors:AddConnector(); separator = "\n">
  <InsertBlank(component.tasks)>
  <component.tasks:AddAtomicModel(); separator = "\n">
  <InsertBlank(component.processes)>
  <component.processes:AddCoupledModel(); separator = "\n">
  <InsertBlank(component.couplings)>
  <component.couplings:AddCoupling(); separator = "\n">
 }
}
>>

InsertBlank(it) ::= <<
<if(it)>

<endif>
>>

AddImport(it) ::= <<
import <it>;
>>

AddConnector(it) ::= <<
ViewableAtomic _<it.name> = new <it.name>();
add(_<it.name>);
```

```
>>

AddAtomicModel(it) ::= <<
ViewableAtomic _<it.name> = new <it.name>(0);
add(_<it.name>);
>>

AddCoupledModel(it) ::= <<
ViewableDigraph _<it.name> = new <it.name>();
add(_<it.name>);
>>

AddCoupling(it) ::= <<
addCoupling(<it.sourceModelName>, "<it.sourcePortName>", <it.
   targetModelName>, "<it.targetPortName>");
>>
```

Listing B.2: The Content of the Process.stg Template File of the DEVS-IM Framework.

```
Process(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
   Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import java.awt.*;
import view.modeling.ViewableAtomic;
import view.modeling.ViewableDigraph;
import view.modeling.ViewableComponent;
import InteractionModel.core.Process;
<component.imports:AddImport(); separator = "\n">

/*
 * id: <component.id>
 * componentType: <component.componentType>
 * description: <component.description>
 */
public class <component.name> extends Process {
 public <component.name>() {
  super("<component.name>");
  <InsertBlank(component.inputs)>
  <component.inputs:AddInputPort(); separator = "\n">
  <InsertBlank(component.outputs)>
  <component.outputs:AddOutputPort(); separator = "\n">
  <InsertBlank(component.tasks)>
  <component.tasks:AddAtomicModel(); separator = "\n">
  <InsertBlank(component.processes)>
  <component.processes:AddCoupledModel(); separator = "\n">
  <InsertBlank(component.couplings)>
  <component.couplings:AddCoupling(); separator = "\n">
 }
}
>>

```

```
InsertBlank(it) ::= <<
<if(it)>

<endif>
>>

AddImport(it) ::= <<
import <it>;
>>

AddInputPort(it) ::= <<
addInport("<it.name>"); // {id: <it.id>, description: <it.
    description>}
>>

AddOutputPort(it) ::= <<
addOutport("<it.name>"); // {id: <it.id>, description: <it.
    description>}
>>

AddAtomicModel(it) ::= <<
ViewableAtomic _<it.name> = new <it.name>(0);
add(_<it.name>);
>>

AddCoupledModel(it) ::= <<
ViewableDigraph _<it.name> = new <it.name>();
add(_<it.name>);
>>

AddCoupling(it) ::= <<
addCoupling(<it.sourceModelName>, "<it.sourcePortName>", <it.
    targetModelName>, "<it.targetPortName>");
>>
```

Listing B.3: The Content of the Task.stg Template File of the DEVS-IM Framework.

```
Task(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
    Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import InteractionModel.core.Task;
import model.modeling.message;

/*
 * id: <component.id>
 * componentType: <component.componentType>
 * description: <component.description>
 */
public class <component.name> extends Task {
 public <component.name>(double activeTime) {
  super("<component.name>", activeTime);

```

```
  <component.inputs:AddInputPort(); separator = "\n">

  <component.outputs:AddOutputPort(); separator = "\n">
 }

 @Override
 protected void perform(message inputMsg, message outputMsg) {
  // TODO Auto-generated method stub

 }
}
>>

AddInputPort(it) ::= <<
addInport("<it.name>"); // {id: <it.id>, description: <it.
   description>}
>>

AddOutputPort(it) ::= <<
addOutport("<it.name>"); // {id: <it.id>, description: <it.
   description>}
>>
```

Listing B.4: The Content of the TransientInputConnector.stg Template File of the DEVS-IM Framework.

```
TransientInputConnector(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
   Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import InteractionModel.core.TransientInputConnector;

/*
 * id: <component.id>
 * componentType: <component.componentType>
 * connectorType: <component.connectorType>
 * description: <component.description>
 */
public class <component.name> extends TransientInputConnector {
 public <component.name>() {
  super("<component.name>");
 }
}
>>
```

Listing B.5: The Content of the TransientOutputConnector.stg Template File of the DEVS-IM Framework.

```
TransientOutputConnector(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
   Simulation)
 *  Version    : DEVS-IM 1.0
```

```
 *  Date       : <date>
 */
package <packageName>;

import InteractionModel.core.TransientOutputConnector;

/*
 * id: <component.id>
 * componentType: <component.componentType>
 * connectorType: <component.connectorType>
 * description: <component.description>
 */
public class <component.name> extends TransientOutputConnector {
 public <component.name>() {
  super("<component.name>");
 }

 @Override
 protected void setDestinations() {
  <component.functions:AddFunction(); separator = "\n">
 }
}
>>

AddFunction(it) ::= <<
addDestination(new <it>());
>>
```

Listing B.6: The Content of the CallOutputConnector.stg Template File of the DEVS-IM Framework.

```
CallOutputConnector(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
    Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import InteractionModel.core.CallOutputConnector;

/*
 * id: <component.id>
 * componentType: <component.componentType>
 * connectorType: <component.connectorType>
 * description: <component.description>
 */
public class <component.name> extends CallOutputConnector {
 public <component.name>() {
  super("<component.name>");
  <if(component.function)>
  setDestination(new <component.function>());
  <endif>
 }
}
>>
```

Listing B.7: The Content of the QueueOutputConnector.stg Template File of the DEVS-IM Framework.

```
QueueOutputConnector(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
   Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import InteractionModel.core.QueueOutputConnector;

/*
 * id: <component.id>
 * componentType: <component.componentType>
 * connectorType: <component.connectorType>
 * description: <component.description>
 */
public class <component.name> extends QueueOutputConnector {
 public <component.name>() {
  super("<component.name>");
 }
}
>>
```

Listing B.8: The Content of the System.stg template File of the DEVS-IM Framework.

```
System(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
   Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import InteractionModel.core.System;
<component.imports:AddImport(); separator = "\n">

/*
 * id: <component.id>
 * description: <component.description>
 */
public class <component.name> extends System {
 public <component.name>() {
  super("<component.name>");
  <InsertBlank(component.components)>
  <component.components:AddComponent(); separator = "\n">
 }
}
>>

InsertBlank(it) ::= <<
<if(it)>

```

```
<endif>
>>

AddImport(it) ::= <<
import <it>;
>>

AddComponent(it) ::= <<
addComponent(new <it.name>()); // {id: <it.id>, description: <it.
   description>}
>>
```

Listing B.9: The Content of the Component.stg Template File of the DEVS-IM Framework.

```
Component(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
   Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import InteractionModel.core.Component;
<component.imports:AddImport(); separator = "\n">

/*
 * id: <component.id>
 * description: <component.description>
 */
public class <component.name> extends Component {
 public <component.name>() {
  <addSuperClass(component)>
  <InsertBlank(component.components)>
  <component.components:AddComponent(); separator = "\n">
  <InsertBlank(component.functions)>
  <component.functions:AddFunction(); separator = "\n">
 }
}
>>

InsertBlank(it) ::= <<
<if(it)>

<endif>
>>

addSuperClass(it) ::= <<
<if(it.parentId)>
super("<component.name>", Long.valueOf(<component.parentId>));<else>
super("<component.name>", null);<endif>
>>

AddImport(it) ::= <<
import <it>;
>>

```

```
AddComponent(it) ::= <<
addComponent(new <it.name>()); // {id: <it.id>, description: <it.
   description>}
>>

AddFunction(it) ::= <<
addFunction(new <it.name>()); // {id: <it.id>, description: <it.
   description>}
>>
```

Listing B.10: The Content of the Function.stg Template File of the DEVS-IM Framework.

```
Function(packageName, component, date) ::= <<
/*
 *  Author     : ACIMS(Arizona Center for Integrative Modeling &
   Simulation)
 *  Version    : DEVS-IM 1.0
 *  Date       : <date>
 */
package <packageName>;

import GenCol.entity;
import InteractionModel.core.Function;

/*
 * id: <component.id>
 * description: <component.description>
 */
public class <component.name> extends Function {
 public <component.name>() {
  super("<component.name>");
 }

 @Override
 public entity exec(entity param) {
  // TODO Auto-generated method stub

  return null;
 }
}
>>
```