Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

CONSTRUCTING HIERARCHICAL MODULAR MODELS IN ALTERNATIVE AND INTERCHANGEABLE REPRESENTATIONS

Hessam S. Sarjoughian[1] and Sheetal C. Mohite[2]

[1] School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA
[2] Lab 126, Amazon, Bellevue, WA, USA

ABSTRACT

Modelers should be empowered to develop component-based models incrementally and iteratively using different modalities. They should not be required to work with an entire model hierarchy or be constrained to pre-defined sub-hierarchy levels that can impose limitations on the visual development of models. This paper introduces a novel visual modeling capability where modelers can work with any sub-hierarchy with any finite number of arbitrary levels and branches. The structural representation of the components with their input/output relationships conforms to the system-theoretic DEVS models and broadly Reactive Modules. Additionally, it is advantageous for the visual and logical aspects of models to be stored and retrieved in databases. Visual models can be generated from their logical counterparts stored in the NoSQL database. A framework supporting a unified logical, visual, and persistent modeling is developed. The framework aids modelers in constructing structural models at scale for simulation and Digital Twins.

1 INTRODUCTION

Modeling is integral to developing digital twins. Models built for simulations generally have logical structural and behavioral specifications that capture various aspects of systems. Developing and changing such specifications incrementally and iteratively is crucial in view of multi-resolution and multi-perspective simulation (Bigelow and Davis 2003). Modelers should be able to develop and analyze composite model hierarchies visually as if they were to use mathematical formulas or programming languages. Visual modeling frameworks enable modelers to develop hierarchical models. Visual models facilitate better understanding and development than mathematical forms and program code counterparts. Furthermore, a framework that can support persistent storage is useful for the systematic evolution and reusability of models, either as visual representation or programming code. Using databases can also reduce challenges associated with model scale and complexity (Sarjoughian 2017). This contrasts with storing models in repositories such as (COMSES, 2008).

This paper details the Persistent and Visual Modeler (PVM) framework, a new hierarchical component-based model aimed at simulation and broadly Digital Twins (Grieves & Vickers, 2017). Modelers can visually construct complex hierarchical model structures while navigating their sub-hierarchies, which can have any desired lower and upper levels according to the principles of modular models composed using input/output ports and connectors. These models adhere to system-theoretic modeling (Wymore, 2018) and Reactive modules (Alur and Henzinger 1999) (De Alfaro and Henzinger 2001). The PVM framework supports the Parallel DEVS formalism (Zeigler et al. 2018) for developing hierarchical models using primitive and composite models. The modularity principle in system theory lends the PVM framework for continuous-time and discrete-time models by allowing any atomic and coupled model to have its output used as input. Models' logical and visual aspects are stored in the NoSQL MongoDB database. Stored models can be seamlessly exported to human-readable JSON syntax. Furthermore, databases can be a key enabler for model analytics and generating new models using Machine Learning. The PVM tool's support

for standalone and service-oriented computing platforms should contribute to the multi-modal hierarchical model development lifecycle. Consequently, the PVM framework contributes to the separation of concerns (i.e., visual, logical, and persistent model modalities) on the one hand and their unified purpose on the other.

2 BACKGROUND

A visual hierarchical component-based model is a tree structure with components and relationships. It has a finite number of levels and branches, each having a finite number of components (nodes) and relationships (part-of) as exemplified in Figure 1(a). Therefore, every hierarchical model has a finite number of unique sub-hierarchies, and every component must be distinct. A sub-hierarchy is a hierarchy section with lower and upper bound levels. It can have one or more branches. It is noted that relationships do not have names and can be uniquely identified using the source and target nodes that connect them. Considering every component to be either primitive (non-decomposable) or composite as required for system-theoretic (Wymore 2018) and reactive modules (Alur and Henzinger 1999) modeling theories, their relationships must conform to a set of rules, such as a composite component not knowing what its encapsulated components are, components interacting via their input and output ports, and components can be uniquely identified as highlighted in node C4 (see Figure 1(b)). Anylogic® (Grigoryev 2018), Astah (Astah 2024), CoSMoS (Fard and Sarjoughian 2015), Decycle (Becker 2019), Degraph (Schauder 2012), and Simulink® (Chaturvedi 2017) are used to examine their visual modeling capabilities (Mohite 2023).

The visual syntax of every sub-hierarchy has a well-defined contiguous boundary. This allows for separating different parts of a hierarchy where every component is connected to at least one other component, assuming the component contributes to the whole model. For multiple sub-hierarchies, it is necessary to guarantee they are consistent with one another in their logical and graphical specifications. For example, if a component is used in a sub-hierarchy, every change made to it and/or its relationships must be mirrored in every sub-hierarchy that has the component. Such a capability for visual modeling is useful as it allows model development with the guarantee that all lower-level sub-hierarchies included in different higher-level (sub-) hierarchies have identical visual forms except the number of their layers displayed or hidden. Therefore, the entire hierarchical model always remains syntactically and visually correct for every modular, component-based modeling language at all times.

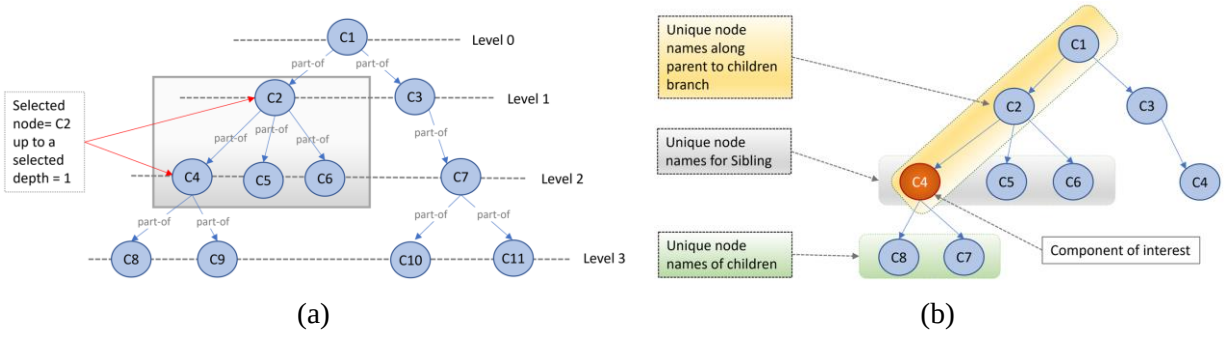


Figure 1: (a) A sub-hierarchy model. (b) Unique naming across branches and levels.

The model hierarchy illustrated in Figure 1 can be enriched using System Theory and Reactive Modules. The methods' logical syntax and semantics must be guaranteed to have correct graphical and persistent modalities. Mathematical theories for continuous-time, discrete-time, discrete-event, and Reactive Modules are widely available (Alur 2015) (Zeigler et al. 2018). The visual nodes have input and output ports with assigned arbitrary variable types and values. Also, the relationships between any two nodes and across nodes in the hierarchy conform to formal specifications. Each relationship has a direction where values without any alteration at a given instance of time on a port from a source node are

instantaneously transmitted to a target node. The logical specification assigned to component-based dynamical systems is lacking in visual tools such as Degraph and Decycle.

3 RELATED WORKS

Many frameworks and tools have been developed for code-based construction of component-based simulation models based on modeling formalisms. A variety of frameworks and tools exist that can be used to develop component-based models graphically. Examples are Anylogic®, CD++ (Bonaventura et al. 2013), CoSMoS, and Simulink®. The hierarchical structure model Main, shown in Figure 2(a), is graphically drawn and used as an exemplar. Anylogic® and Simulink®, among others, are used to develop the example model.

Simulink® is widely used for developing causal block diagrams with components, ports, and connectors. It has a general-purpose graphical editor for developing models, each of which has input and output ports. A composite component encapsulates components. The composite and its encapsulated components are connected using input and output ports. Structurally, composite DEVS and Simulink® models are indistinguishable from one another.

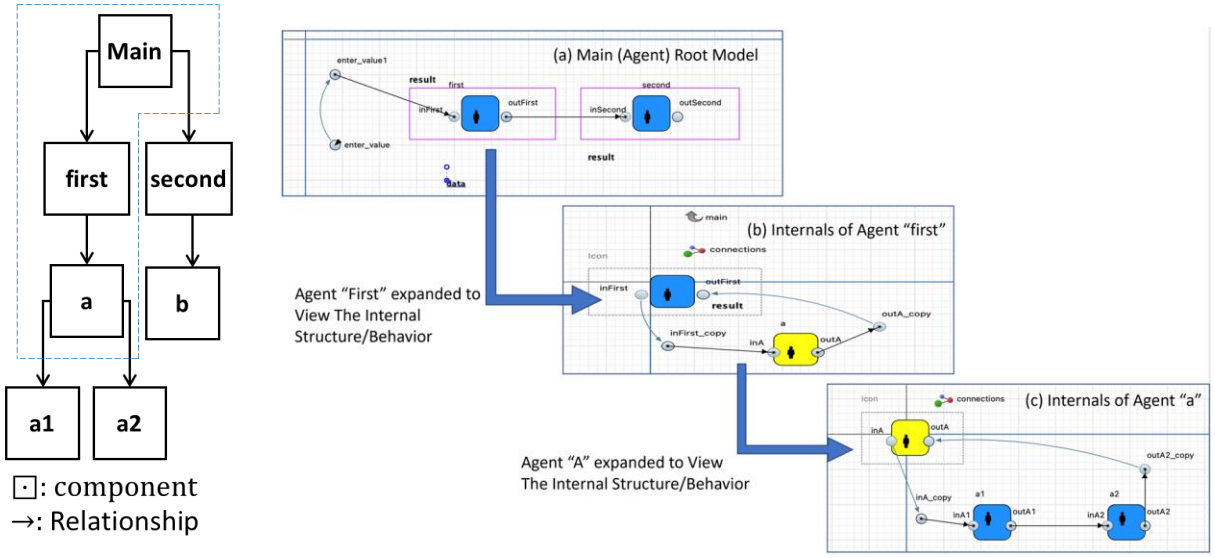


Figure 2: (a) An exemplar tree-structure model drawn as boxes and arrows. (b) A section of the exemplar model developed in AnyLogic®.

In contrast to Simulink®, AnyLogic® supports modeling agents that should share data with one another; therefore, agents are not strictly modular. An agent's parameters and variables exposed to other agents can be defined using shared variables to capture the concepts for ports and connectors. The *first* and *second* are defined as agents and coupled in the Main which serves at the root of the model hierarchy (see Figure 2(b)). The agent *first* uses the *inFirst* and *outFirst* public variables serving as input and output port names. The agent *a* is embedded in the *first* agent. The public variables of the two embedded agents must be shared to act as a connector. The variables must be configured as "dependent" to accept input from some agent. Unlike Simulink®, two dependent variables cannot be connected unless the additional variable *inFirst_copy* is introduced between *inFirst* and *inA*. The examination of AnyLogic® illustrates the simplicity and power of the component abstraction and modularity afforded by system theory and Reactive Modules in view of visual modeling.

UML Component modeling lends itself to visually developing prototypical component-based models. A model is defined as components that can have ports and interfaces. The components can be connected to develop hierarchical composite models (Booch et al. 2007). The component diagrams offer a rich set of features targeting software design specifications but are not targeted for developing simulation models. UML component diagrams can be developed using professional tools such as Astah (Astah 2024). Table 1 shows the comparison of the AnyLogic®, Simulink®, and Astah. Within the scope of this paper, other aspects of these frameworks and tools, such as model-to-code generation (or vice versa), model execution, and UI configuration, are excluded.

The PVM is based on CoSMoS (Component-based Modeler and Simulator) framework which loosely separates logical, visual, and persistent models but also unifies them. There are two important essential differences between PVM and CoSMoS. The former has the novel concept of capturing any change in logical model in every sub-hierarchy visualization. The latter supports collaborative model development (Sarjoughian and Vignesh 2010) (Sarjoughian et al. 2011) (Fard and Sarjoughian 2015). The length and height of components with their coordinates are automatically calculated and may not be changed. Connectors are automatically drawn and cannot be modified. The concept of storing models in databases is the same in these frameworks, except the PVM uses RESTful-enabled MongoDB while CoSMoS uses Microsoft Access in a standalone fashion. Also, in contrast to PVM, CoSMoS supports automatic management for developing alternative models of a system with rules for the unique naming of components and their instances using the concept of template, instance template, and instance model classification.

Table 1: Visual component-based modeling frameworks.

Feature	Description	Astah	AnyLogic®	Simulink®	CoSMoS	PVM
Modeling	Hierarchical tree structures	Y	Y	Y	Y	Y
	Component-based	Y	Y	Y	Y	Y
	Component reusability	Y	Y	Y	Y	Y
	Dynamic UI configuration	Y	Y	Y	Y	Y
	Sub-hierarchies with arbitrary levels	N	N	N	N	Y
	Structural metrics	N	N	N	Y	Y
	State machines	Y	Y	Y	Y	N
	Unified visual, logical, and persistent models	N	N	N	Y	Y
Storage	Storage and retrieval of models in a database	N	N	N	Y	Y

Considering component-based model development, to our knowledge, visual modeling with arbitrary lower/upper hierarchy levels is not supported in any other component-based modeling frameworks, including AnyLogic®, CD++, CoSMoS, and Simulink® as well as UML/SysML modeling tools such as Astah. A comparison of the above tools focusing on visual modeling is shown in Table 1. It should be noted this comparison is neither intended to consider (subtle) differences nor to be complete. The focus is on visual component-based model development. For example, components can be strictly modular, as defined in system theory, and reactive modules can be object-based or object-oriented classes. AnyLogic® is not strictly modular compared to others. It can be used to develop hierarchical component diagrams separate from class diagrams with graph structures with different relationship types (e.g., dependency and inheritance). These tools also support different visual syntaxes. Similarly, state machines can have simple states with transitions or hierarchical states with time-constrained transitions. These tools support different means for storing and retrieving models (flat files vs. database) and support standalone and distributed computing platforms.

4 MODELING FRAMEWORK

As noted in the previous section, component-based modeling frameworks are widely used in model-based design and simulation. A system is defined as a composite model that has smaller, self-contained components. This simple concept becomes quite powerful when it is formulated in terms of complementary mathematical, graphical, and storage formulations. An implementation of the PVM concept and framework is detailed below (Mohite 2023).

4.1 Model Development Framework

The concept for the modeling framework is shown in Figure 3. The framework's Model-View-Control (MVC) design pattern supports the distinct visual, persistent, and logical aspects of input/output component-based model development. The visual modeling layer provides the user with an interface for incremental and iterative creation of individual and hierarchical models. The persistent modeling layer uses MongoDB to store and retrieve models' logical and visual aspects (Fard and Sarjoughian 2021). The logical modeling layer uses elementary concepts and theories for modeling (Zeigler et al. 2018) and software design for reactive modules (Alur 2015). All components can be assigned input and output variables (ports). Every composite component has one or more connectors (couplings). These logical models have only structural specifications. The semantics of the ports and connectors (i.e., couplings) between any two components conform to the Parallel DEVS formalism, which requires the consumption of inputs and the generation of outputs by every component to occur independently at chronological time instances. Transfers of inputs and outputs for input-to-input, output-to-input, and output-to-output connectors occur synchronously. The visual and persistent structural models strictly conform to the Parallel DEVS specification. That is, no component can have any connectors between any of its inputs and outputs. These structural rules are satisfied by every visual and every persistent model.

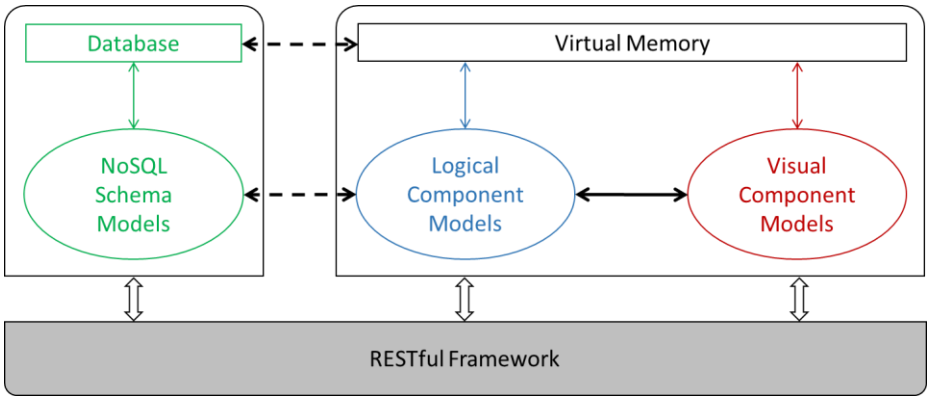


Figure 3: An illustration of the PVM architecture.

The framework supports creating models and adding components, ports, and couplings along with the capability to modify or delete any part models. The framework's architecture accounts for functional and non-functional requirements. The former refers to the specifications of services that the system must offer, the appropriate responses the system must generate to specific inputs, and the expected behavior of the system. The latter refers to the services or functions such as visual representations of components and their compositions. The visual and persistent modeling capabilities formulated below adhere to the logical structural specification for developing classical/parallel DEVS models and Reactive Modules for a/synchronous software design. The PVM framework uses MongoDB in the WEAP-KIB-LEAP RESTful framework (Fard and Sarjoughian 2021). Models in the database can be exported to JSON and code for the

DEVS-Suite simulator (ACIMS 2023a). The structural metrics of every model or any part can be obtained and visualized.

4.2 Model construction

The model development involves creating and adding components, ports, and couplings. The common scenario for the model creation is that the modeler wants to create a new model graphically. As in commonly used modeling tools, the modeler interacts with a visual environment that has Editor and Browser panels and drop-down menus. As outlined below, the Editor and Browser panels support constructing diagrams for model components with all their elements. The common scenarios are to create a model, add a new component as a subcomponent in a selected component from the hierarchy, delete a selected component, or navigate to a component's sub-hierarchy.

- Create new models: User interacts with the UI to create new whole (root) models while guaranteeing their uniqueness. This model can have inputs and outputs.
- Create component: User adds the component created whole models. The new component is created with the default one input and one output port.
- Create port: User can select any component and add a finite number of input/output ports.
- Constraints: Independent uniqueness for the input and output port names for each component is enforced. The uniqueness of component names for every branch in the model hierarchy is enforced.
- Rename components and ports: User can change the name of any component or any port.
- Resize component: User can resize the dimensions of any component and change its coordinate.
- Redraw coupling: User can redraw couplings using a finite number of anchor points to reconstruct hierarchical models that are simpler to visualize.
- Detach and move: User can change the hierarchy level of any component by detaching it and moving it to another model hierarchy.
- Delete components: User can select and delete any part of the root model. All components and ports with their couplings are removed.
- Delete port: User can select and delete any port. All couplings that use any deleted port are deleted.
- Delete couplings: The modeler can delete ports without deleting the connecting ports.

4.3 Hierarchical modeling

The model development detailed in Section 4.2 focused on constructing components as standalone elements. The PVM Browser panel can be used to view the hierarchical structure of the model. Modelers can perform basic operations from the model hierarchy. They can perform other operations, including coupling components encapsulated in composite components. Common scenarios for the hierarchy operations are to add a new component as a subcomponent in a selected component from the hierarchy, delete a selected component, and navigate a certain section of the hierarchy from a selected component.

- Hierarchy levels: user navigates and selects a desired component. The modeler chooses the number of levels to open the sub-hierarchy in a new tab with the name of the selected component.
- Add component: user adds the component from the Browser or Editor panel. The component is created and added to the desired component (as a sub-component).
- Create coupling: user selects the ports of the desired components and connects them in the Editor panel. The ports are validated.
- Delete component: user selects and deletes a component to delete from the Browser or Editor panel. The ports and couplings associated with this component are deleted.

4.4 Model storage and export

The framework provides a database for storing, exporting, and retrieving models. These capabilities work hand-in-glove with visual modeling. For incremental and iterative model development, persistent storage support is required. The capabilities are outlined next.

- Store model: user can store the logical model elements along with their counterpart visual elements and layouts. Every stored model is validated for its uniqueness in the database repository.
- Save model: user can incrementally save model elements logical and visual layouts.
- Load model: user can select and load (retrieve) any existing model from the database.
- Export model: user can export any model in JSON format to any local or remote location.

4.5 Operation modes

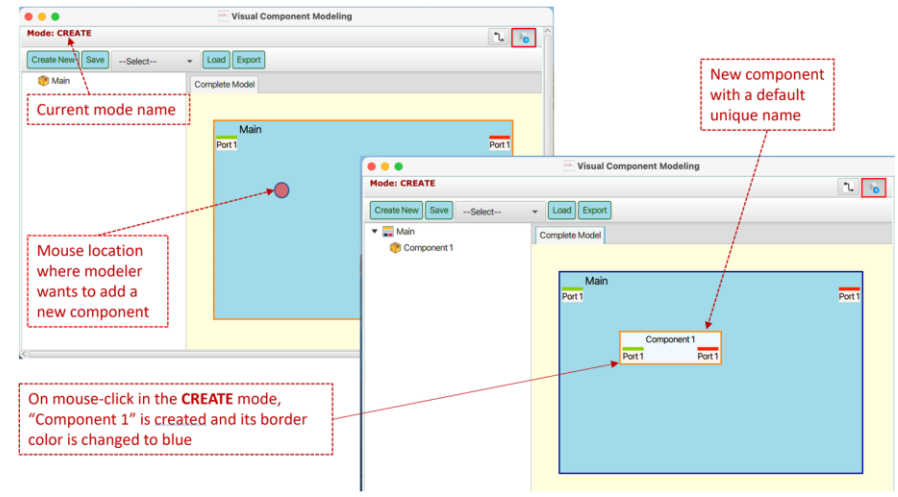
To make a clear distinction between the various mouse actions and provide a user-friendly interface the user modes have been defined in the UI. There are three user interaction modes.

- Create: users can perform only creation functionalities. In this mode, users can use the mouse events to create components with ports at a desired position in the Editor Panel. The input and output port positions are automated.
- Change: users can perform the update operations on the nodes such as component and port. Since deletion or detach/attach of the component changes the model layout, the update can also be carried out in the change mode.
- Coupling: users need to select two desired ports for a straight line to be drawn between them. Thus, to keep the UI understandable and user-friendly, this mode is introduced specifically for couplings. The coupling-specific operations such as creating, deleting, or changing a straight line to a polyline can happen in this mode.

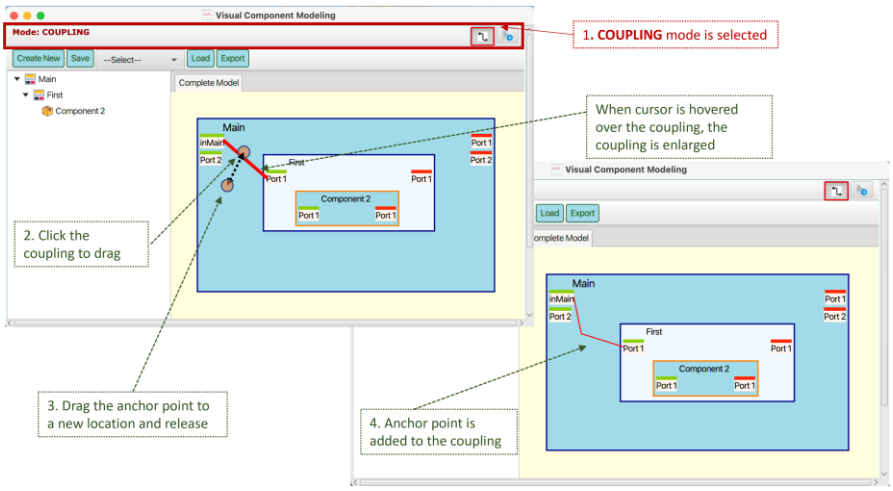
Table 2 shows the UI features according to the *CREATE*, *CHANGE*, and *COUPLING* modes. They are mainly associated with the creation, update, and deletion of operations for the components, ports, and couplings, as well as making changes to the layout of the components and couplings. The model's logical and database aspects are not directly associated with any of the user modes. The concept of mode plays a key role in delineating different kinds of operations to work seamlessly from the vantage point of logical and persistent modeling aspects. Defining components and ports are separated from defining couplings. The change mode is used to modify the visual models. The models in the database are updated to ensure the visual and persistent models have the same exact structures as the ones in the database.

Table 2: Operation modes for model development.

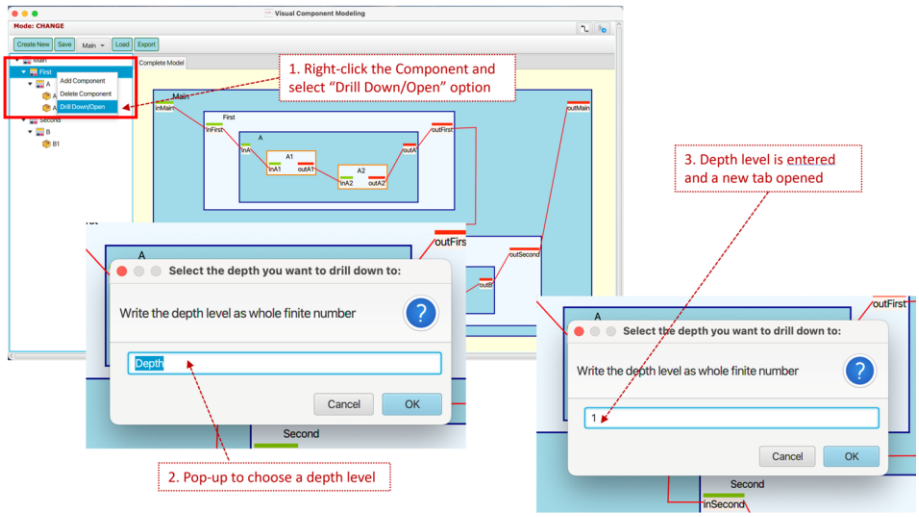
Operations	CREATE	CHANGE	COUPLING
Create Components and Ports	Y	N	N
Resize/Move Components	N	Y	Y
Rename/Detach/Attach/Delete Components and ports	N	Y	N
Create/Redraw couplings	N	N	Y



A root component model named **Main** is created in the *Create* mode. The border color of the model is orange because no component is encapsulated in it. **Component 1** is encapsulated in the **Main**, making it a composite component with its border color changed to blue.



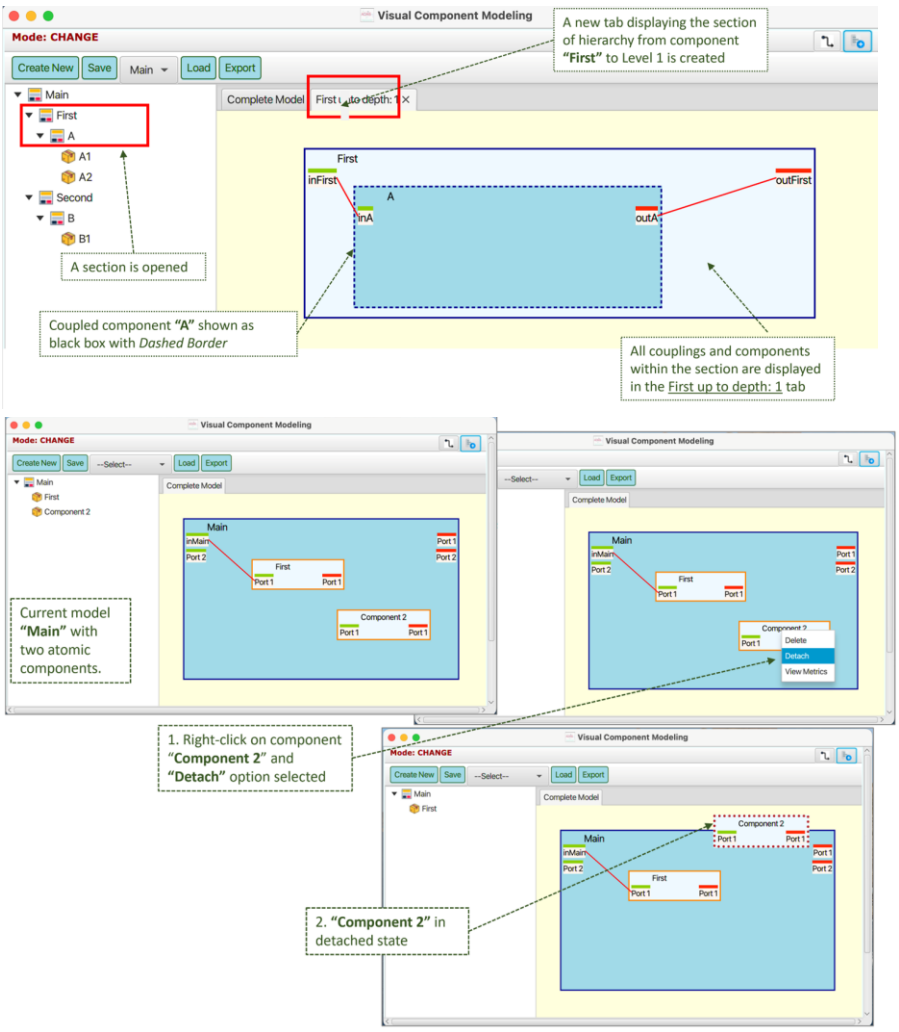
Creating a coupling from the **Main** component to its encapsulated **First** component in the *Coupling* mode. Redrawing the coupling using anchor points.



Creating a sub-hierarchy with its own Editor panel supported with the Edit, Create, and Coupling modes.

Figure 4: Exemplified Operations in the CREATE and COUPLING modes.

The operations closely related to the construction of structural hierarchical models of sub-hierarchies with making changes to them are illustrated in Figures 4 and 5. The operations for storing and retrieving models from the MongoDB are excluded. The layout of primitive and composite components, ports, and couplings and their coloring scheme are configurable by the modeler (ACIMS 2023b). Visual models can be constructed from models created in JSON format and stored in databases. Every model's component tree-structure hierarchy, external input, internal, and external output coupling types are verified throughout the model development to satisfy the DEVS formalism (Sarjoughian and Vignesh 2010).



A sub-hierarchy is selected for the model **First**. It is shown in a new tab named *First up to depth: 1*. Detaching model **Component 2** from model **First** and placing in a new sub-hierarchy tab.

Model **Component 2** is detached. Its solid borderline changes to a dashed line border with a maroon color. **Component 2** can be attached to any existing model.

Figure 5: Detaching/attaching components in the CHANGE mode.

5 DISCUSSION

Creating models for a system is an inherently demanding undertaking, particularly when its structure and behavior should be defined in detail as a collection of components and relationships. Models of such dynamical systems are challenging to specify unless some amount of complexity is abstracted out and/or scale is reduced. These kinds of models serve simulation purposes well (analysis and design) as they focus on specific questions to be formulated and answered. On the one hand, some models do not have to have

realistic complexity and scales (e.g., drones). On the other hand, other models are developed as cellular automata that may have millions of parts (e.g., cancer biology).

Digital Twins, however, are expected to mirror actual systems, acting as replicas and as closely as possible executed in real-time (Liu et al. 2021). Developing digital twin models using component-based visual modeling can quickly become impractical when there are many hundreds or more components and relationships. Considering Digital Twins (and simulations), the concept shown in Figure 3 and the PVM framework are aimed at contributing to managing model complexity and scale measured in terms of the number of components, their I/O, and hierarchical relationships. Modelers can choose any portion of a model and change the details of any component (more or fewer components encapsulated inside it) and its relationships. As described above, graphical and program code with persistent database should lead to commercial and research frameworks better equipped to meet the needs of Digital Twins. For tools such as Simulink®.

The methodology presented in the paper should lend itself to Simulink® and others such as Architecture Analysis and Design Language (AADL) (Hudak and Feiler 2007) with its implementation of Open Source AADL Tool Environment (OSATE) (OSATE 2016) that conforms to the class of component-based, modular, hierarchical modeling methods. For frameworks that are based on the class of object-based and object-orientation concepts and programming languages such as Analogic, the graphical modeling method may be possible with strong encapsulation and input/output modularity. Supporting model persistence using databases should be feasible.

6 CONCLUSION

The Persistent Visual Modeler (PVM) framework is designed to create hierarchical models that consist of components organized into levels and branches. It supports various types of models—logical, visual, and persistent—helping users explore and understand different aspects of their models. Unlike other component-based modeling frameworks that are limited to two or all levels, the PVM framework allows modelers to define and develop sub-hierarchies with any number of levels and branches.

The framework is evaluated against industry and academic tools used for simulation development. As Digital Twins are expected to grow significantly in the coming years, modelers should benefit from the PVM framework's modularity across logical, visual, and persistent models. The PVM framework simplifies the creation of large, complex models by separating logical and visual abstractions from their database storage. Its use of RESTful services and databases supports incremental and iterative model development at scale and reuse by multiple modelers.

The principles of the PVM framework developed for component models could also simplify and scale the development of behavioral state machines and activity models. Furthermore, the PVM framework has the potential for extensions, such as code generation directly from visual models, model reuse across multiple databases, and collaborative modeling.

ACKNOWLEDGMENTS

This research is partially supported under the NSF Grant #CNS-1639227. We thank the reviewers for their critiques and suggestions.

A VISUAL DESIGN SPECIFICATION

Figure 6 illustrates the UML class diagrams for some of the parts of the PVM framework that closely relate to visual model development and storage in MongoDB database (Mohite 2023).

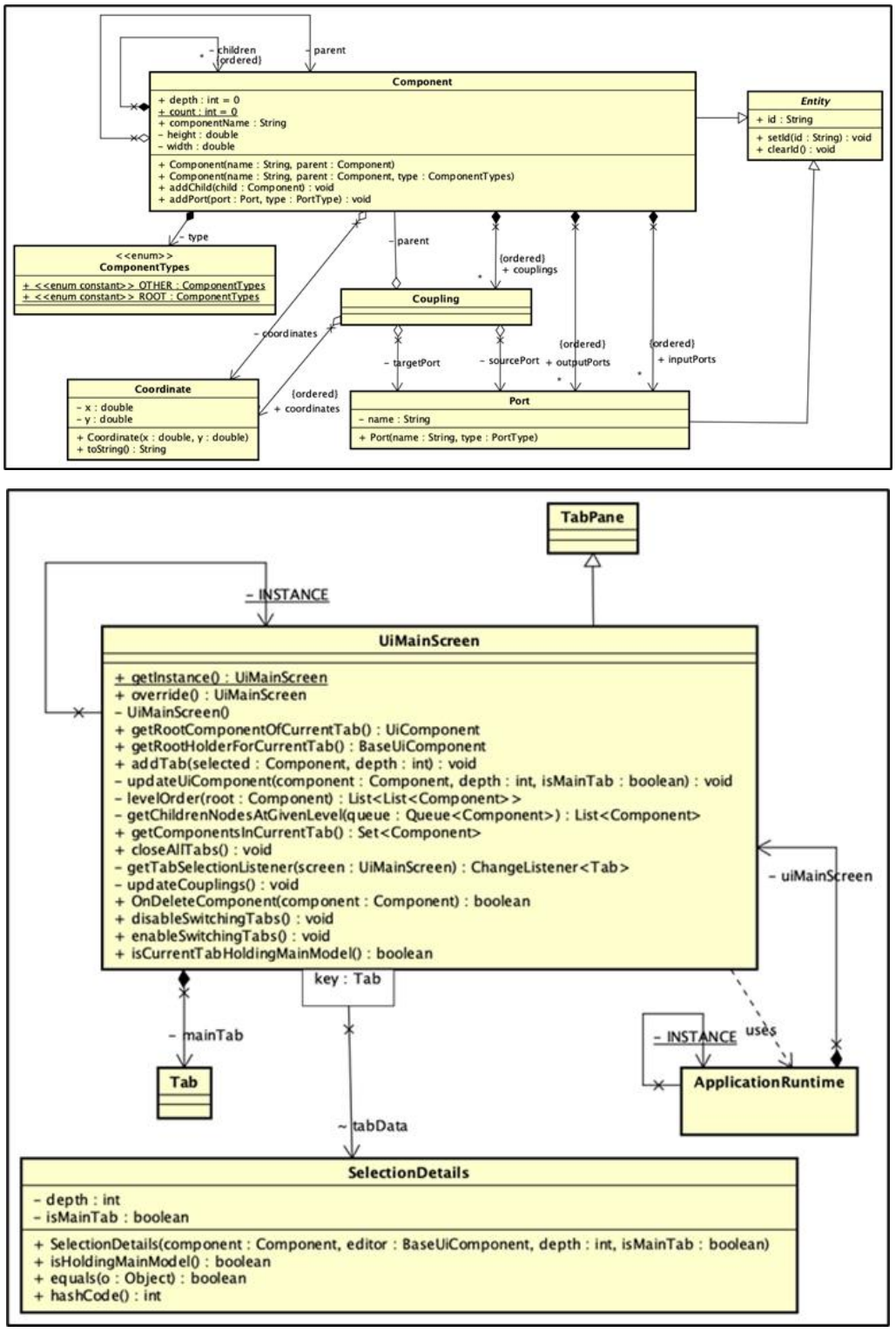


Figure 6: UML class diagram for specifying visual (i) components' coordinates and (ii) operation modes for creating, changing, and couplings.

7 REFERENCES

ACIMS. (2023a). *DEVS-Suite Simulator*. Retrieved from Arizona Center for Integrative Modeling & Simulation: https://acims.asu.edu/devs-suite/.

ACIMS. (2023b). Persistent and Visual Modeler. *Arizona Center for Integrative Modeling and Simulation*. Retrieved from https://acims.asu.edu/persistent-visual-modeler/.

Alshareef, A. and Sarjoughian, H. S. (2021). Hierarchical Activity-Based Models for Control Flows in Parallel Discrete Event System Specification Simulation Models. *IEEE Access, 9*, 80970 – 80985.

Alur, R. (2015). *Principles of cyber-physical systems.* MIT Press.

Alur, R. and Henzinger, T. A. (1999). Reactive modules. *Formal methods in system design, 15*, 7-48.

Astah. (2024). Retrieved from https:/astah.net.

Becker, O. (2019, October 04). *Decycle*. Retrieved March 29, 2023, from https://github.com/obecker/decycle/

Bigelow, J. H. and Davis, P. K. (2003). *Implications for model validation of multiresolution, multiperspective modeling and exploratory analysis.* Santa Monica, CA: RAND Corporation.

Bonaventura, M., Wainer, G. A., and Castro, R. (2013). Graphical modeling and simulation of discrete-event systems with CD++ Builder. *Simulation: Transactions of The Society for Modeling and Simulation International, 89*, 4-27.

Booch, G. R. (2007). *Object-oriented analysis and design with applications* (Third edition ed.). Addison-Wesley.

Chaturvedi, D. K. (2017). *Modeling and simulation of systems using MATLAB and Simulink.* CRC Press. Retrieved from https://www.mathworks.com/help/pdf_doc/simulink/simulink_gs.pdf.

COMSES. (2008). *Computational Modeling in the Social and Ecological Sciences*. Retrieved June 15, 2024, from https://www.comses.net/.

De Alfaro, L. and Henzinger, T. A. (2001). Interface Automata. *ACM SIGSOFT Software Engineering Notes, 26*(5), 109-120.

Fard, M. D. and Sarjoughian, H. S. (2015). Visual and persistence behavior modeling for DEVS in CoSMoS. *SpringSim (TMS-DEVS)* (pp. 227-234). IEEE Press.

Fard, M. D. and Sarjoughian, H. S. (2021). A RESTful Persistent DEVS-Based Interaction Model for the Componentized WEAP and LEAP RESTful Frameworks. *Winter Simulation Conference*, (pp. 1-12).

Grieves, M. and Vickers, J. (2017). Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems. In J. F. Kahlen, *Transdisciplinary Perspectives on Complex Systems* (pp. 85–113). Springer.

Grigoryev, I. (2018). *AnyLogic 8 in Three Days: A Quick Course in Simulation Modeling.* (Sixth, Ed.)

Hudak, J. a. (2007). *Developing AADL models for control systems: A practitioner's guide.* Carnegie Mellon University, Software Engineering Institute.

Liu, M., Fang, S., Long, H., and Xu, C. (2021). Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems, 58*, 346-361.

Mohite, S. C. (2023). *A Unified Visual and Persistent RESTful Tool for Modular and Hierarchical Modeling.* Arizona State University, Tempe, Arizona.

OSATE. (2016). *Open Source AADL Tool Environment*. Retrieved September 17, 2023, from https://osate.org/.

Sarjoughian, H. S. (2017). Restraining complexity and scale traits for component-based simulation models. *Winter Simulation Conference* (pp. 675-689). IEEE.

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

Sarjoughian, H., Nutaro, J., and Joshi, G. (2011). Collaborative Component-based System Modeling", Journal of Simulation, Vol. 5, No. 2, 77-88. *Journal of Simulation, 5*(2), 77-88.

Schauder, J. (2012, December 28). *Degraph*. Retrieved April 21, 2023, from http://riy.github.io/degraph/index.html.

Wymore, W. A. (2018). *Model-based systems engineering.* CRC Press.

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

AUTHOR BIOGRAPHIES

HESSAM S. SARJOUGHIAN is an Associate Professor of Computer Science and Computer Engineering in the School of Computing and Augmented Intelligence (SCAI) at Arizona State University (ASU), Tempe, Arizona. His research interests include model theory, poly-formalism modeling, collaborative modeling, simulation for complexity science, and M&S frameworks/tools. He is the co-director of the Arizona Center for Integrative Modeling and Simulation (https://acims.asu.edu). He can be reached at hessam.sarjoughian@asu.edu.

SHEETAL C. MOHITE is a software engineer at Amazon Lab 126. She earned her Computer Science MS degree in 2023 from the School of Computing and Augmented Intelligence (SCAI) at Arizona State University (ASU), Tempe, Arizona. She can be reached at sheetal041292@gmail.com.