

Activity Specification for Time-based Discrete Event Simulation Models

by

Abdurrahman Alshareef

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

Approved August 2019 by the
Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
Georgios Fainekos
Joohyung Lee
Ming Zhao

ARIZONA STATE UNIVERSITY

December 2019

ABSTRACT

Computational models for relatively complex systems are subject to many difficulties, among which is the ability for the models to be discretely understandable and applicable to specific problem types and their solutions. This demands the specification of a dynamic system as a collection of models, including metamodels. In this context, new modeling approaches and tools can help provide a richer understanding and, therefore, the development of sophisticated behavior in system dynamics. From this vantage point, an activity specification is proposed as a modeling approach based on a time-based discrete event system abstraction. Such models are founded upon set-theoretic principles and methods for modeling and simulation with the intent of making them subject to specific and profound questions for user-defined experiments.

Because developing models is becoming more time-consuming and expensive, some research has focused on the acquisition of concrete means targeted at the early stages of component-based system analysis and design. The model-driven architecture (MDA) framework provides some means for the behavioral modeling of discrete systems. The development of models can benefit from simplifications and elaborations enabled by the MDA meta-layers, which is essential for managing model complexity. Although metamodels pose difficulties, especially for developing complex behavior, as opposed to structure, they are advantageous and complementary to formal models and concrete implementations in programming languages.

The developed approach is focused on action and control concepts across the MDA meta-layers and is proposed for the parallel Discrete Event System Specification (P-DEVS) formalism. The Unified Modeling Language (UML) activity meta-models are used with syntax and semantics that conform to the DEVS formalism and its execution protocol. The notions of the DEVS component and state are used together according to their underlying system-theoretic foundation. A prototype tool support-

ing activity modeling was developed to demonstrate the degree to which action-based behavior can be modeled using the MDA and DEVS. The parallel DEVS, as a formal approach, supports identifying the semantics of the UML activities. Another prototype was developed to create activity models and support their execution with the DEVS-Suite simulator, and a set of prototypical multiprocessor architecture model specifications were designed, simulated, and analyzed.

To My Mother, Haya bint Nasser

ACKNOWLEDGMENTS

I wish to thank the members of my graduate supervisory committee for their service. I am especially thankful for my faculty advisor and committee chair, Hessam Sarjoughian, for his guidance and support. I also appreciate the fruitful discussion and valuable insights that have been put through by the committee members Georgios Fainekos, Joohyung Lee, and Ming Zhao. Their valuable inputs have certainly helped in shaping the outcome and formulating the contribution with further clarifications regarding related formalisms and the logic foundation of this work.

I would also like to thank my colleagues at the Arizona Center for Integrative Modeling and Simulation. The members and visiting scholars of this lab have been outstanding in countless discussions over different topics, research interests, and problems. I also appreciate the School of Computing, Informatics, and Decision Systems Engineering and, notably, the Computer Science Program for being an excellent academic environment for learning and conducting research. I would certainly go beyond the space limit if I were to name all faculties and talented staff whom I greatly benefited from throughout my years at the School and the Program.

A special appreciation goes sincerely beyond all academic credits to my brother Turki, my sisters Nora, Asma, and Arwa, and all of my family for their awe-inspiring spirits. I have been a longtime recipient of their tremendous encouragement and unlimited support, without which I might have given up such a challenging pursuit at many time points.

Undertaking this Ph.D. was supported by a scholarship provided by King Saud University.

TABLE OF CONTENTS

| | Page |
|---|------|
| LIST OF TABLES | xi |
| LIST OF FIGURES | xii |
| CHAPTER | |
| 1 INTRODUCTION | 1 |
| 1.1 Problem Statements and Attempted Works | 2 |
| 1.1.1 Model Ambiguity and Implicit Assumption | 2 |
| 1.1.2 Code Generation and Execution | 3 |
| 1.1.3 Complexity and Scale | 5 |
| 1.1.4 Classification and Meta-Layers | 6 |
| 1.1.5 Model Evaluation and Architecture Selection | 6 |
| 1.1.6 Temporal Structure | 7 |
| 1.2 Related Work and Contribution | 8 |
| 1.3 Published Works | 13 |
| 1.4 Preliminary Notations | 14 |
| 2 BACKGROUND | 16 |
| 2.1 Metamodeling and the DEVS Formalism | 17 |
| 2.1.1 MDA and Model Layers | 17 |
| 2.1.2 DEVS Atomic Model | 18 |
| 2.1.3 EMF-DEVS Atomic Model | 19 |
| 2.2 Foundational UML Subset | 20 |
| 2.2.1 UML Activities | 20 |
| 2.2.2 Abstract Syntax | 21 |
| 2.2.3 Execution Model | 22 |
| 2.2.4 Foundational Model Library and Base Semantics | 23 |

| CHAPTER | Page |
|---|------|
| 2.3 Formalisms and Languages for Discrete Event Modeling | 23 |
| 2.3.1 Parallel DEVS Atomic Model | 24 |
| 2.3.2 Statecharts | 26 |
| 2.3.3 Activities and Actions | 27 |
| 2.4 Eclipse Modeling Framework | 29 |
| 3 BEHAVIORAL DEVS METAMODELING | 31 |
| 3.1 Related Work | 34 |
| 3.2 Atomic DEVS Metamodeling | 36 |
| 3.2.1 Meta-Behavior Modeling in EMF | 38 |
| 3.2.2 Constrained Meta-behavior Modeling | 42 |
| 3.3 A Processor Example Behavioral Metamodel Snippet | 44 |
| 3.4 Conclusion | 45 |
| 4 AN APPROACH FOR ACTIVITY-BASED DEVS MODEL SPECIFI- CATION | 49 |
| 4.1 Related Work | 51 |
| 4.2 Approach | 53 |
| 4.2.1 Three Views for Specifying Atomic Model Behavior | 55 |
| 4.2.2 Activity Specifications for Atomic DEVS Model | 57 |
| 4.2.3 Action Specifications for Atomic DEVS Model | 58 |
| 4.3 Statecharts and Activities | 60 |
| 4.4 Conclusion and Future Work | 62 |
| 5 DEVS SPECIFICATION FOR MODELING AND SIMULATION OF THE UML ACTIVITIES | 65 |
| 5.1 Related Work | 67 |

| CHAPTER | Page |
|---------|--|
| 5.2 | Activities Simulation Through DEVS: Finding Rigor 69 |
| 5.2.1 | A DEVS Grounding for UML Activities 70 |
| 5.2.2 | The Semantics of Activities 72 |
| 5.3 | Network Switch: an Example 75 |
| 5.3.1 | Modularity 76 |
| 5.3.2 | The Generality of the Models 78 |
| 5.4 | Future Work 79 |
| 5.5 | Conclusion 80 |
| 6 | ACTIVITY-BASED DEVS MODELING 82 |
| 6.1 | Related Work 84 |
| 6.2 | Activity-based DEVS Modeling Approach 86 |
| 6.2.1 | Categorizing the Activity Specification 89 |
| 6.2.2 | Note on Coupled Models and Behavioral Specification 90 |
| 6.2.3 | Controlled Coupling Using Activities Control Nodes 91 |
| 6.3 | EMF-based Modeling Engine 95 |
| 6.3.1 | Activity-based DEVS Ecore 95 |
| 6.3.2 | Activities Graphical Definition and Tooling 98 |
| 6.3.3 | Mapping 100 |
| 6.3.4 | Preliminaries on the Validation of the Activity-based DEVS Models 102 |
| 6.4 | Activity-based Modeling for Multiple Input Processor with Queue . . 103 |
| 6.4.1 | Interpreting the Processor Model in the DEVS-Suite Simulator 105 |
| 6.5 | Conclusion 108 |
| 7 | PARALLELISM SEMANTICS IN MODELING ACTIVITIES 110 |

| CHAPTER | Page |
|---------|--|
| 7.1 | The Role of Action in Activities and Other Behavioral Models 112 |
| 7.1.1 | Atomic Model and Action 113 |
| 7.1.2 | State and Time for Actions 115 |
| 7.2 | Multiprocessor Architectures 116 |
| 7.3 | Parallelism Semantics 119 |
| 7.3.1 | Multiple Branching via Split Nodes 121 |
| 7.3.2 | Joining Multiple Paths and Interruptions 124 |
| 7.3.3 | Using Control Nodes for Job Coordination 125 |
| 7.4 | Related Work 127 |
| 7.5 | Conclusion 129 |
| 8 | MODEL-DRIVEN TIME-ACCURATE DEVS-BASED APPROACHES FOR CPS DESIGN 131 |
| 8.1 | Background 134 |
| 8.1.1 | Parallel DEVS 135 |
| 8.1.2 | Real-Time DEVS (RT-DEVS) 135 |
| 8.1.3 | Action-Level Real-Time DEVS (ALRT-DEVS) 135 |
| 8.1.4 | Timed Automata 137 |
| 8.2 | Related Work 137 |
| 8.3 | Action-Level DEVS Specification Using Activity Modeling 139 |
| 8.3.1 | CPS Activities Metamodel 139 |
| 8.3.2 | The Modeling and Simulation of a Traffic Intersection 140 |
| 8.4 | Interacting with Reactive Computational-Physical Systems 144 |
| 8.5 | Verification of the CPS Activities Models 145 |
| 8.5.1 | Reasoning About Temporal Behavior 147 |

| CHAPTER | Page |
|--|------|
| 8.6 Conclusion | 148 |
| 9 METAMODELING ACTIVITIES FOR HIERARCHICAL COMPONENT- BASED MODELS | 150 |
| 9.1 Component-based Modeling | 152 |
| 9.2 Coordinating Between Server Components | 156 |
| 9.3 The Specification of Action and Control in Activities | 160 |
| 9.3.1 Coordinator Statecharts | 163 |
| 9.3.2 Constructing Hierarchy within Activities | 164 |
| 9.4 Demonstrating with Activity Modeling Tool | 166 |
| 9.5 Related Work | 170 |
| 9.6 Conclusion | 174 |
| 10 ACTIVITY SPECIFICATION FOR TIME-BASED DISCRETE EVENT SIMULATION MODELS | 176 |
| 10.1 On Simulation Modeling Architectures and Frameworks | 178 |
| 10.1.1 Modeling Layers | 178 |
| 10.1.2 Related Work | 180 |
| 10.2 DEVS Specifications for Activity Nodes | 182 |
| 10.2.1 Mapping UML Activity Control, Object, and Flow to DEVS Model, Port, and Coupling | 186 |
| 10.3 Exploiting Parallelism | 191 |
| 10.3.1 Parallelism Semantics | 192 |
| 10.3.2 Simple Experiment for an Archetype Divide and Conquer Architecture in DEVS-Suite Simulator | 194 |
| 10.4 Flow Selection Schemes | 196 |

| CHAPTER | Page |
|---|------|
| 10.4.1 A Pipeline Architecture | 198 |
| 10.4.2 A Multi-Server Architecture | 200 |
| 10.5 Framework for Activity Modeling and Simulation | 201 |
| 10.5.1 Time for Activities | 202 |
| 10.5.2 Observations of Temporal Analysis with Activities | 204 |
| 10.5.3 Simulating Activities in DEVS-Suite | 209 |
| 10.6 Conclusion | 212 |
| REFERENCES | 214 |
| APPENDIX | |
| A OTHER CONTRIBUTIONS | 224 |
| A.1 Infusing Simulatability into Software Models | 225 |
| A.1.1 Transforming Activity Models to DEVS Models: Autonomous Vehicles | 225 |
| A.2 Toward Precise Semantics of Actions | 226 |
| A.2.1 Introduction | 227 |
| A.2.2 The Atomic Model and the Action | 227 |
| A.2.3 A Processor Model | 228 |

LIST OF TABLES

| Table | Page |
|--|------|
| 5.1 A Subset of the Mapping for Activity Elements | 73 |
| 6.1 Activity Specifications for Atomic DEVS Model | 88 |
| 6.2 Decision and Merge Node Figure Definition | 101 |
| 7.1 A Subset of Activities Elements and Briefly Their Semantics with Re- spect to Parallelism in Correspondence with DEVS | 120 |
| A.1 A Set of Atomic and Coupled Models for Activities DEVS Modeling and Simulation..... | 225 |

LIST OF FIGURES

| Figure | Page |
|--|------|
| 1.1 Related Works and Background Map Highlighting Our Activity Modeling Approach. | 10 |
| 1.2 Notation Examples of Essential Activity Modeling Elements Along with Their Treatment as Components with Accounts to Multiple Ports and Couplings. | 14 |
| 1.3 Activity Modeling for a Wymore (1993) Server System. | 15 |
| 3.1 From Mathematical to UML to EMF Modeling. | 39 |
| 3.2 A Metamodel for Atomic DEVS Model with State Transitions. | 43 |
| 3.3 Ecore for a Processor with Primary State Transitions for the External Transition Function. | 46 |
| 4.1 A Subset of Behavior Elements and their Relationships In UML 2.5 Metamodel. | 54 |
| 4.2 Different Views of Activities DEVS Modeling. | 55 |
| 4.3 Activity Models for the Processor (Simplified). | 60 |
| 4.4 The Overall View of the Approach and Relationships between Different Models of Consumer Behavior. | 63 |
| 5.1 A Simplified View of Employing Concepts in M&S for Activities Modeling. | 69 |
| 5.2 The Action, Which Is a Special Type of Activity Node, Is Treated as an Atomic Model with Some Input and Output Ports. | 71 |
| 5.3 The Network Switch Parallel DEVS Coupled Model. | 76 |
| 5.4 An Activity for the Network Switch Coupled Model. | 76 |
| 5.5 A Simulation View for the High-Level Activity Constructs Used to Model Network Switch (Implemented in DEVS-Suite). | 78 |

| Figure | Page |
|---|------|
| 6.1 An Activity for Synchronizing Outputs from the Generators Prior to Processing. The Simulation View (Right) Is for the Corresponding Implementation in DEVS-Suite. The Join Node as an Atomic Model Is in <i>Waiting</i> Phase to Synchronize the Input through the Other Port from the Second Generator..... | 92 |
| 6.2 Activity-based DEVS Metamodel..... | 97 |
| 6.3 Visual Canvas for Activity-based DEVS Modeling. | 99 |
| 6.4 Activity-based DEVS Modeling for Multiple Input Processor with Queue. | 106 |
| 7.1 A Classification for Formal and Semi-Formal Component-Based Modeling Approaches with Respect to Structure and Behavior. | 111 |
| 7.2 A View of an Action and State. | 116 |
| 7.3 We Examine Different Abstractions for Different Architectures. The Component Views with Couplings Are Shown in the Top, and Their Corresponding Activities Are Shown in the Bottom. The Areas in Grey Highlight the Control Nodes That Are Used to Represent the Coordinating Procedure in Each Architecture. The Letter P Stands for Processor and A for Action. In (A), the Coordinator Is Represented by the <i>Decision</i> Node D to Direct the Job According to Some Condition Associated with the Outgoing Flow. Conditions Are Visually Omitted. In (B), the Job Is Either Brought Back to the <i>Decision</i> Node d1 to Be Directed Again, or Sent out If Completed. In (C), the Job Is Divided in the <i>Split</i> Node S and Combined Back in the <i>Join</i> Node J after Processing. | 118 |
| 7.4 A View of the Architectures and Some of Their Corresponding Behaviors. | 128 |

| Figure | Page |
|--------|---|
| 7.4 | A View of the Architectures and Some of their Corresponding Behaviors.129 |
| 8.1 | Actions in the CPS Are Characterized into Four Types. the Types in Grey Are Crucial from a CPS Standpoint since They Are Akin to the Tight Coupling between Cyber and Physical Parts. Actuating Actions, for Example, Can Impact the Physical Environment Directly and Therefore Their Consequences Are Critical. 134 |
| 8.2 | The Activities Metamodel Is Circumscribed and Extended with CPS Action. ALRT-DEVS Metamodel Is Also Linked with the Activities Metamodel at a High Level to Establish the Grounding for the DEVS Modeling and Simulation of the CPS Activity. The X, Y, and S Sets Are the Same as Those Defined for P-DEVS. Some Cardinalities Are Visually Omitted. The Elements with Italic Are Abstract Super-Type Elements. 141 |
| 8.3 | The Activity for Modeling Traffic Intersection and Simulating It In DEVS-Suite..... 142 |
| 8.4 | Phase Trajectories for Different Scenarios for <i>Toggle</i> as a CPS Action. . 146 |
| 9.1 | Modeling the Dual Server System in CoSMoS. 154 |
| 9.2 | An Activity for the External Transition Function of the Coordinator. . . 158 |
| 9.3 | Activity of a Multi-Sever Archetype Architecture Is Devised Using Various Activity Constructs. S_1 and S_2 Actions Represent the Jobs Service. C_1 and C_2 Represent Conditions for Choosing Flow Directions. The Nodes inside the Dashed Line Area Highlight the Role of the Activity Control Elements in the Manipulation of the I/O Flow. 163 |
| 9.4 | Modeling the Coordinator Statecharts in CoSMoS. 164 |

| Figure | Page |
|---|------|
| 9.5 Hierarchical Construction with Activities..... | 166 |
| 9.6 A Metamodel for Hierarchical Activities Developed Using Ecore. | 167 |
| 9.7 Viewpoint Specification in Sirius. | 168 |
| 9.7 Viewpoint Specification in Sirius. | 169 |
| 9.8 Modeling Multi-Server Activity in the Developed Activity Modeling Tool..... | 170 |
| 9.8 Modeling Multi-Server Activity in the Developed Activity Modeling Tool..... | 171 |
| 9.9 The Simulation View of the Developed Activity for the Multi-Server System after the Code Generation for DEVS-Suite Simulator. | 172 |
| 10.1 Illustration of the Mapping of Different Activity Nodes with Accounts to Multiple Ports and Couplings..... | 190 |
| 10.2 Activity-Based Modeling of the Divide and Conquer Architecture. | 193 |
| 10.3 The Trajectories for the State Variable <i>phase</i> and the Input <i>in</i> and Output <i>out</i> Ports With Events for the a_2 Component. | 197 |
| 10.4 Different Abstractions of the Pipeline Architecture with Possibly Dif- ferent Temporal Attributions in Their Simulations. | 199 |
| 10.5 Activity-Based Modeling of the Multi-Server Architecture. | 201 |
| 10.6 A High-Level Sketch Illustrating (A) the Incorporation of Action and Control Node on the One Hand and State on the Other, and (B) a Conceptual Relationship between I/O and Activity Pin. | 202 |

| | | |
|------|---|-----|
| 10.8 | Throughput Is Observed by Simulating the Activity of Divide and Conquer in DEVS-Suite, Given Different Numbers of Actions and Tasks Arriving at the Same Time. T_{pt} Is Equal to 10 Time Units in All Cases, and T_c Is Assigned Linearly Relative to the Number of Actions, Where a Greater Number of Actions Requires a Greater T_c Value. | 212 |
| A.1 | The Integration of the New Packages Within the Current Architecture. | 226 |
| A.2 | The Multiple Views for Modeling and Simulation of an Intersection. . . . | 226 |
| A.3 | The Action Abstraction Is Situated at the Heart of Many Behavioral Specifications and Thus Used As a Bridge Between the Formal Specification and Other Semi-Formal or Informal Modeling Approaches. | 227 |

Chapter 1

INTRODUCTION

Computational models for complex systems are subject to many difficulties, among which is the ability to be directly applicable to various essential needs and natural phenomena. In recent years, significant advances in developing dynamic models for systems have used a variety of model abstractions under the model-driven architecture and model-based design umbrella of semi-formal methods. Fundamental to these efforts is the metamodeling concept spanning component-based structural and behavioral model specifications. From this vantage point, we have proposed an activity specification to be developed based on the discrete event modeling approach with the accounts to time notion. Models of this nature evolve with the intent of posing and answering specific and profound questions according to general set-theoretic model specifications with supporting simulation execution protocols.

Since the modeling process tends to be time-consuming and highly expensive, we focused on facilitating some concrete means that aid formulating both the structures and behaviors of models simultaneously. The aim is to develop models that lend themselves to a higher degree of rigor during the early stages of the modeling and simulation life cycle. Intermediary abstractions, afforded through metamodeling, become indispensable for exploring uncharted territories of model spaces in ways by which their computational aspects expand and potentially lead to discovering new models.

Introducing a new concept at a meta-layer necessitates substantial efforts at concrete layers to realize the concept in attempts to pursue its benefits. Such realization comes into play while recognizing, at the fundamental level, the fact that achieving

a consistent formal system with full provability is out of reach. Furthermore, an increase in system complexity demands richer abstractions and boundaries to be more realizable in a useful way. Such models lead to simulations that can be more useful for the understanding and predictability of intertwined time-based behavioral dynamics. The subject of this dissertation is highly intrinsic to the systems to be modeled and dependent on the modeler's ability. Basic research in this topic is needed to help mitigate some of the fundamental barriers that arise when developing higher quality system architecture and design specifications. Such designs, once executed, can provide useful insights much earlier than is currently possible.

1.1 Problem Statements and Attempted Works

Particular problems have been the subjects of the research conducted for this dissertation. In the following, we introduce them with some remarks about the research that was carried out by formulating the problems and relevant concepts. We discuss more details throughout the remaining chapters, but first, we highlight general views about some of the issues of interest and the efforts made to address them.

1.1.1 Model Ambiguity and Implicit Assumption

Models are generally treated as simplifications and, therefore, contain fewer details about interests in their targeted domains. The process of simplification used in some modeling approaches have led to ambiguities. For example, some aspects of the Unified Modeling Language (UML) are known to be challenging to use. The problem also becomes visible concerning particular properties that are implicit or otherwise arbitrary. Modelers using such modeling languages often face the burden of navigating through incomplete or possibly contradictory abstractions. These limitations result in having models that do not lend themselves to a sound framework. For the context

of a sound framework, models should not only have well-defined syntax and semantics but also be able to correctly simulate or execute model behaviors.

We employ concepts like actions and control in behavioral metamodels to provide a means for understanding behavioral aspects of a system under study. We argue that establishing a rigorous mathematical grounding for a strictly selected subset of the UML is necessary to achieve essential benefits concerning execution. We also suggest that discrete event modeling frameworks can serve as suitable candidates. Thus, we propose formalizing the activity modeling via a set of system theory atomic and coupled models as defined in the Discrete Event System Specification (DEVS). Therefore, foundational elements of activities and actions are modeled and mapped into a set of atomic and coupled models where they can collectively serve as a basis for grounding different diagrams via coupling with the entry-level capability of modeling and simulation for activities.

1.1.2 Code Generation and Execution

Code generation is a well-recognized problem that has been the target of tremendous efforts and a variety of proposed frameworks and techniques to work around it. While it is difficult to have a fundamental solution to this problem, code generation frameworks continue to grow to accelerate designs ranging from embedded systems to highly networked systems. Tackling such a problem can take many different forms. The resulting programs may also encounter other issues in terms of interpretations. When it comes to a model, the distinction between its automated code generation and interpretation plays an essential role since each has a different set of artifacts. The code generation produces artifacts that, in turn, can be subject to further modification and optimization. Interpretation provides some understanding of the separation of concerns such as model continuity with loose dependency on code

generation targeted for specific programming languages. The problem becomes more challenging with increasing dependencies that can arise quite easily in systems that have intrinsic complexity and scale traits. Even with the widely used abstraction concepts and frameworks (e.g., Eclipse Papyrus) and the basic types of relationships (e.g., dependency), code generation is restrictive.

Some frameworks, such as the Model Driven Architecture, have been proposed to support the creation of simulation models. The overarching role of the frameworks has been to aid model specifications in a disciplined fashion, but simulation is not the primary focus. They can, however, provide intermediary layers from the higher mathematical models to their corresponding software specifications from a mostly structural standpoint. Unlike structural modeling, behavioral modeling is known to be more difficult, particularly when functional operations require non-trivial control schemes. This observation has resulted in proposing activity-based behavior modeling for simulation. We consider actions as the fundamental units of behavioral modeling and its use alongside state-based models. The state-based and flow-based abstractions can serve the complementary roles needed for developing rich component-based modeling and simulation frameworks.

In addition, we have developed a prototype modeling engine that demonstrates key aspects of the proposed activity modeling approach. The engine is produced using the Eclipse Modeling Framework along with tools supporting graphical model development and code generation for simulation. We also detail the relevant aspects of the created metamodel in terms of modeling and simulation. A large number of the activity artifacts from the vantage point of DEVS behavioral modeling, including actions and control, are covered in detail. We also discuss the semantics of the artifacts for time-accurate requirements for simulation.

1.1.3 Complexity and Scale

Models are developed to sustain, and therefore, having mechanisms for a longer lifetime is of the utmost importance. Different kinds of measures can help in evaluating model specifications. Component-based models, for example, are measured in terms of the number of components and relationships they have. Quantitative measures are usually more straightforward; however, interpretations of such measures differ across domains. In contrast, qualitative measures, such as model reuse, are harder to define. The significance of some relationships or components as opposed to others among application domains varies. Models may have intrinsically different characteristics that complicate the scale and complexity measures, especially those that are tied to model behavior. A family of models can help to deal with the complexity and scale issues, which can become overwhelming when there is only one model.

In discrete system modeling (Zeigler *et al.*, 2018b), the exact and approximate scales for the structures of modular, hierarchical models are measured in a straightforward fashion. Many real systems, such as enterprise processes, are known to have large scales and complexities. Such systems have a large number of components with and even more significant number of relationships. The scale and complexity measurements for a system that has a variety of components and connection types are hard to obtain. Structural and behavioral homogeneity in models such as cellular automata and synchronous reactive models lend themselves to simpler scalability and complexity measures. Behavioral model specifications that are grounded in component-level action and state abstractions are more likely to have scale and complexity measurements.

1.1.4 *Classification and Meta-Layers*

The problem of classification continues to be the subject of extensive study, including efforts such as set theory conceptualizations. Aside from being subjective depending on, for example, data types and execution semantics, classifications can be brittle, without careful use of abstractions. The problem becomes relatively straightforward if it is to be worked out in an ad hoc manner or when a system has rigid behavior. Conversely, it is possibly a conundrum when a particular and specific rigor is necessary to deal with system complexity.

We proposed a set theory specification in the context of DEVS. A mapping takes the elements of the DEVS to their counterparts that conform to MDA. The idea is to examine the capability of developing platform-independent models that can transform into platform-specific models. We shed light on and introduce behavioral metamodeling for discrete event simulation models. We also discuss the behavioral specification from the standpoint of the MDA framework with a three-layer model abstraction consisting of the metamodel, concrete model, and instance model. Prototypes were developed to describe the three-layer modeling approach from the perspective of DEVS and realized in the Eclipse Modeling Framework. A behavioral metamodel expands to the core model of the framework, and afterward, we examine it while considering other metamodels for supporting structural features. Furthermore, we discuss some observations regarding behavioral metamodeling, model validation, and code generation.

1.1.5 *Model Evaluation and Architecture Selection*

It is not common practice to develop models for subject systems in conjunction with the environment within which they are expected to execute. Therefore, models

of experiments can play a significant role in evaluating the effectiveness of the subject model, particularly from the standpoint of the behavioral dynamics. Software modeling methods that explicitly account for state and actions can lend themselves to this purpose. Evaluating models of time-critical and safety-critical systems is necessary. Even though some results can be rendered through standalone simulations of a subject system, evaluations supported by experimental models are needed to gain greater insight into the system dynamics. Without such an approach, some aspects of the system may not be possible to evaluate. Therefore, activity modeling can help in developing richer models for both the subject and experiment models.

1.1.6 Temporal Structure

There is a growing need for systems to account for the notion of time, which is necessary for accommodating time-sensitive behaviors. Although essential in many domains, the mere inclusion of time expands the state space of such dynamical systems with far more complications. Different theories and models suggest different representations and calculations of time, and they vary significantly in definition from those as simple as the linear temporal logic to as complex as a full account of time as an abstract quantity with real values. Such notions are essential, especially when dealing with well-known challenging problems in computing, such as parallelization and synchronization.

To further the proposed activity approach for modeling and simulation, we examine it further concerning parallelization and synchronization of the data and control flows. A time base, regardless of its granularity, is explicitly necessary to account for parallelism in a simulation environment. We examine that by dissecting the basic temporal properties that are related to control constructs within activity modeling in the viewpoint of the parallel DEVS formalism and its time base.

Performance analysis and verification of cyber-physical systems (CPS) are good examples when it comes to time sensitivity in decision-making processes. The interaction between computational and physical parts is of particular interest in modeling such systems. We employ some research on simulation and model-checking for designing computational-physical interactions in the context of a basic CPS and propose an action-level model-driven activity modeling approach based on DEVS. We employ time intervals (TIs) to govern the communication between computational and physical components at the level of actions. We extend the activities metamodel to instantiate activities suitable for time-critical CPS. We also demonstrate with an example of a vehicular traffic intersection model with verification.

1.2 Related Work and Contribution

Many existing formalisms have been useful for a variety of needs, among them modeling and verification. Common examples of these formalisms are Petri nets (Murata, 1989), timed automata (Alur, 1999), answer set programming (Lifschitz, 1999), and DEVS (Zeigler *et al.*, 2000). Each one can be used to dissect certain aspects of the system state. However, there has not been a formal way that a complete representation of a system can take place except with a large degree of constriction or abstraction. Thus, it becomes necessary to use a variety of formalisms for various needs in a complementary manner. Fundamental difficulties may arise, especially regarding heterogeneity; however, certain guarantees can be afforded if some analysis effort takes place. This fact is behind the high cost of developing models that lend themselves to profound formalisms. Less formal approaches such as UML, System Modeling Language (SysML), and MDA may become appealing with beneficial practices and accompanying frameworks. Environments such as Eclipse Modeling Framework (Steinberg *et al.*, 2008) are useful, but they are prone to complexity is-

sues (Fondement *et al.*, 2013) demanding parallel efforts to withstand a certain degree of rigor and scale.

Despite advances in the degree of coverage of the state space, restricted formalisms such as Petri nets continue to require extensions like time Petri nets (Berthomieu and Diaz, 1991) and high-level Petri nets (Jensen and Rozenberg, 2012). Introducing a continuous-time base in the specification leads to an inherent difficulty in interpreting any classical computational model from a system-theoretic standpoint. This difficulty is evident due to the necessary treatment in such a setting for challenging aspects of heterogeneity, composability, discretization, multiple resolutions, and the likes.

Figure 1.1 depicts a map of the overall literature in this dissertation, along with possible paths and research areas. It includes some efforts along these paths and areas; however, others remain unexamined. The expected capability dictates making certain decision along the way.. Arriving at a certain execution is one capability that some research is going after, while others are verification and simulation. The means of how such capabilities are delivered also varies. Transformation, extension, and formalization are some techniques to account for when conducting efforts that involve constructs with an inadequate syntactical and semantic definitions. We will discuss in more detail some of the research on these three capabilities. Nevertheless, it is a voyage in spaces with multiple paths to arrive, ideally and ultimately, at such capabilities.

Some formalisms account for specifications and are therefore used for modeling particular aspects of systems. The modeling efforts then translate to means in which verification of specific properties can take place under certain conditions. Some properties are reachability (Hwang and Zeigler, 2009), progress (Misra, 2001) or liveness (Lamport, 1989), maximality (Misra, 2001), and safety (Lamport, 1989; Misra, 2001; Alur, 2015). The path toward achieving verification of such properties involves cre-

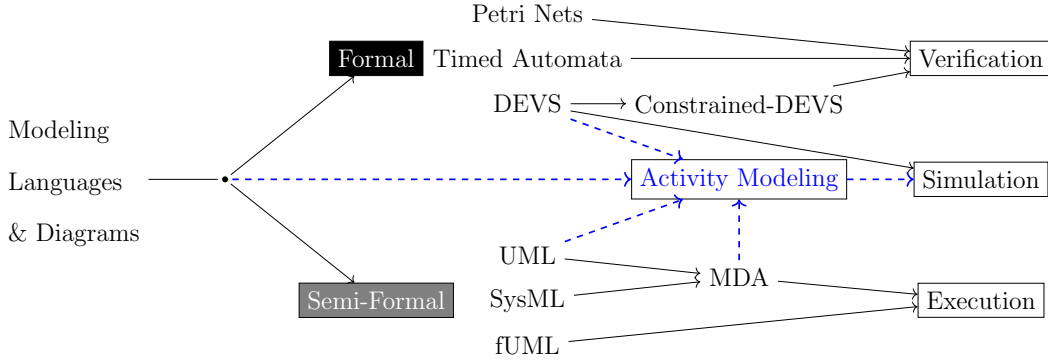


Figure 1.1: Related Works and Background Map Highlighting Our Activity Modeling Approach.

ating counterpart representatives in formalisms like timed automata, Petri nets, or some extensions thereof, such as hybrid I/O automata (Lynch *et al.*, 2003) or high-level Petri nets. Researchers have proposed different mappings to various formalisms or extensions thereof. The degree of coverage in these efforts also varies. Some of them account for basic elements (Rafe and Rahmani, 2008), while others account for a wider set of constructs and definitions (Störrle and Hausmann, 2004), such as the ones defined in the activity metamodel (OMG, 2005). For example, the latter includes detailed treatment for semantics of various activity constructs such as executable node, control nodes, and various patterns of activity edges. The work concludes, however, with some critical remarks about the feasibility of aligning activities to Petri nets and vice versa. We profoundly account for such remarks in this dissertation by establishing the distinction between verification and simulation as two different sought capabilities. The former can be supported by taking the path of Petri nets and some counterpart for it in the DEVS arena such as finit Deterministic DEVS (FD-DEVS) (Hwang and Zeigler, 2009) and constrained-DEVS (Gholami and Sarjoughian, 2017). While the former depends on use of TA, the latter is grounded on extending the DEVS formalism to support verification with the benefit of a full-fledged modeling and sim-

ulation environments (e.g., DEVS-Suite simulator [ACIMS, 2019] with supports for time series and behavior monitoring and debugging).

On the other side, shown in the lower part of Figure 1.1, some existing transformations, and extensions suggest the notion of model execution and define semantics along the way during the model development and through proposed execution engines. Some of these approaches employ the MDA (Miller and Mukerji, 2003) with adapted mechanisms such as model to text (M2T) and Query/View/Transformation (QVT). These mechanisms enable code generation for producing code snippets and programs in target programming languages. An earlier work is executable UML (Mellor *et al.*, 2002). More recently, the foundational subset of UML (fUML) (OMG, 2013) extends the UML with sets of actions with more elaborate semantic definitions and an execution model. It also proposes some mappings to the programming language Java. Such standardization efforts led to the development of execution engines for the UML activity diagram including Moka (Eclipse Foundation, 2016b) and others, in different modeling tools. The mapping is then drawn from the introduced specialization of activity elements (e.g., *Read Self Action*) to their counterpart in the target programming language (e.g., *this* in Java). From a high-level point of view, the relationship between modeling constructs and their code counterpart is apparently one-to-one.

Some works followed through to overcome the problem of clutter that comes up due to such relationships with large graphical notation to represent a relatively simple procedure. Among these is the action language for foundational UML (Alf), a textual language to represent fUML, and a proposal by Bedini *et al.* (2017). Other approaches employ MDA with metamodeling, profiling, and other extension mechanisms to compensate in models of UML and SysML with more concrete details of implementations. One goal is to equip them with better inclination to simulation (Foures *et al.*, 2012) or execution (Mayerhofer *et al.*, 2013). We thoroughly examined

these studies on various occasions, especially the ones that used DEVS, such as Nikolaidou *et al.* (2008), Risco-Martín *et al.* (2009), Cetinkaya *et al.* (2011), and Kapos *et al.* (2014).

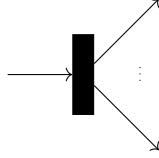
Transformation takes place in particular between the DEVS, on the one hand, and the UML, SysML, or MDA, on the other (e.g., Yonglin *et al.* (2009); Sarjoughian and Markid (2012); Mittal and Martín (2013b)). These studies attempted to holistically look into the problem of a potential mismatch between the formal and semi-formal specifications. The MDA frameworks deliver some benefits for transformation from a primarily structural vantage point, and certain mappings are selective, leaving many details to be abstract given that metamodels are inherently incomplete. Many implementations and manifests take place at concrete layers to compensate and complement as much as possible for their counterpart representation at the higher layers. Lower level implementations are necessary to arrive at some artifacts that facilitate simulation or execution but the correspondence between formal and semi-formal specification is mostly unfulfilled, especially regarding semantic definitions, and it will likely remain as such.

In this dissertation, we attempt to use profound simulations to realize, to some degree, the behavioral specification of certain types of models. We examine an inclusive subset of activity constructs with attention to their syntax and semantics. From a modeling perspective, we employ the notion of the model as defined in modeling formalisms such as discrete event or discrete time system specifications. Models of such a nature encounter a more rigorous and explicit specification of their time base, I/O sets, state sets, and I/O and state segments. The result are models that benefit from basic definitions in general modeling languages and semi-formal methods. Yet, such models lend themselves to the mathematical discrete event system specification (i.e., the DEVS) and its abstract simulator.

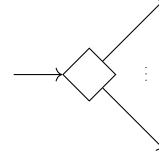
1.3 Published Works

Several publications have preceded the submission of this dissertation. The earlier work by Sarjoughian *et al.* (2015) presented in Chapter 3 has been carried out to set the stage and highlight issues regarding specifying behavior at the meta-layers from more of an observational standpoint. The work was followed up by the activity-based DEVS model specification Alshareef *et al.* (2016) that we present in Chapter 4. The approach was studied further and extended by Alshareef and Sarjoughian (2017) and Alshareef *et al.* (2018), as shown in Chapter 5 and 6, respectively, to cover aspects of coupling and the coupled model as opposed to focusing only on the atomic one. We propose a basic mapping between activities and DEVS (Alshareef and Sarjoughian, 2017). Also, we presented a specification for action at the Ph.D. Colloquium at the Winter Simulation Conference (Alshareef, 2017), which we have included in Appendix A.2. Another demonstration with a traffic example is included in Appendix A.1, which we presented at the Spring Simulation Conference Demo Session 2017.

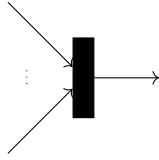
The semantics of activity modeling is discussed with respect to parallelism by Alshareef and Sarjoughian (2018b) in Chapter 7. Chapter 8 includes an example model for CPS design as discussed by Alshareef and Sarjoughian (2018a). We developed this example after extending the metamodel to account for temporal elements, particularly those in action-level real-time DEVS, such as time window. More recent work has also been published to account for the hierarchical construction in the activity (Alshareef and Sarjoughian, 2019), and it is presented in Chapter 9. Work on a journal article with a more comprehensive view is ongoing and included in Chapter 10. I have also contributed to work on a profile for cognitive modeling and domain-specific modeling by visiting scholars to Arizona Center for Integrative Modeling and Simulation (ACIMS), namely Zhu *et al.* (2017, 2018).



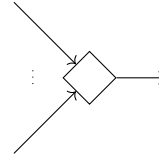
(a) An Example for a *Fork* Node, Where
Outgoing Flows Are Synchronized.



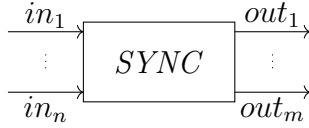
(d) An Example for a *Decision* Node, Where
One Outgoing Flow Is Activated.



(b) An Example for a *Join* Node, Where
Incoming Flows Are Synchronized.



(e) An Example for a *Merge* Node, Where
One Incoming Flow Is Activated.



(c) A Corresponding *Sync* Model for Both
Fork and *Join*.



(f) A Corresponding *Select* Model for Both
Decision and *Merge*.

Figure 1.2: Notation Examples of Essential Activity Modeling Elements Along with Their Treatment as Components with Accounts to Multiple Ports and Couplings.

1.4 Preliminary Notations

In Figure 1.2, we briefly present some of the essential modeling elements that we use throughout this dissertation. The *fork* node is the one that synchronizes dispatching outputs through its outgoing flows. The *join* works similarly but for incoming flows for which it expects an input. Since they are symmetric, we refer to them both as *SYNC*. Similarly, the *merge* and *decision* nodes are used to select one flow for proceeding as with incoming in the former and outgoing in the latter. They are both referred to as *SELECT*.

Figure 1.3 exemplifies the notations by modeling the server system in Wymore

(1993), which is a component of the manufacturing system model. A set of states along with *NOONE* describes this system, which indicates the absence of inputs at some time instant. The system description includes a random input to allow errors to happen during service and therefore determine the result of the service. A next signal transmits in conjunction with the output upon service completion. The description also includes a transition function with a time lapse between the issuance of subsequent signals. The system is described with the queuing system as a way to facilitate deduction about service with more details. The length of the queue, for example, is variable and can be of interest to determine the adequacy of simple models.

An activity for such a system includes two actions *a* and *b*. We developed such an activity using our activity tool. The flow coming through input and service time parameters gets directed toward both actions. An error is injected to only one action where its flow is neglected by the merge node after one action sends its outgoing flow. This example highlights some essential aspects of our approach. The subsequent chapters elaborate on this treatment.

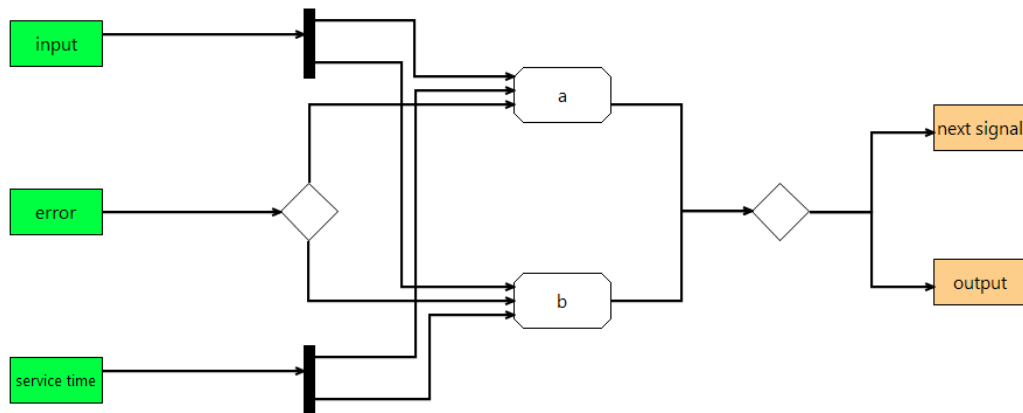


Figure 1.3: Activity Modeling for a Wymore (1993) Server System.

Chapter 2

BACKGROUND

The background about the Discrete Event System Specification (DEVS) (Zeigler *et al.*, 2000) formalism is essential to lay the ground for a computational basis and use to establish a link with other models of computation and discrete event systems. The atomic model is of particular interest in this dissertation due to the inherent difficulty of the behavioral specification, given the level of abstraction of the simulator. This difficulty is realized at the heart of this work and discussed from the vantage point of formalism itself and other existing frameworks and architectures such as the Model Driven Architecture (MDA). In such frameworks, the primary role is to facilitate handling structures.

Due to the behavioral limitations in existing works, we give a brief background about the so-called *Semantics of a Foundational Subset for Executable UML Models* (fUML) (OMG, 2018). A similar initiative has taken place and precedes fUML under the notion of Executable Unified Modeling Language (xUML) Mellor *et al.* (2002). Throughout this dissertation, we look further and examine extensions that are taking place, whether in the proposed subset itself or via the underlying execution engines that are being developed in conjunction with it. We have made arguments about the necessity of simulation in such efforts to shed light and clarify subtleties that might arise due to modeling restrictions and execution.

Another essential background for this dissertation is about modeling discrete event systems. We primarily focus on the DEVS formalism; however, we also benefit from other well-known modeling formalisms such as State-charts. We also discuss the activities and actions in this context as we think of them to complement different ap-

proaches, especially when having some more detailed behavioral models. The Eclipse Modeling Framework (EMF) is used in this dissertation as a realization for the MDA and a concrete means to examine multiple aspects of it, especially its support for the behavioral specification.

2.1 Metamodeling and the DEVS Formalism

In this work, our goal is to develop concepts that can enable building a framework capable of specifying meta-behavior for atomic DEVS models. Then, we use these means to create concrete atomic DEVS models. Toward this goal, we employ Model-Driven Engineering (MDE) and in particular, the MDA framework with its EMF realization. Although there are a variety of DEVS-based modeling and simulation tools, in this work, we use the DEVS-suite simulator for developing the proposed behavioral DEVS metamodel.

2.1.1 MDA and Model Layers

The MDA framework has been proposed for developing software systems (OMG 2003). Its central concept is a four-layer model abstraction hierarchy. A key abstraction concept in MDA is for a classifier and its instances to form a two-layer hierarchy. A classifier has an abstract specification that can have one or more instances. Classifiers are universal and instances are specific. Every classifier is at a higher level of abstraction relative to its instance. Instances are related to one or more classifiers via conformance relationship. The implication is having complementary models, each of which has a specific role to play and collectively provide a disciplined roadmap for developing software systems. Each higher-level layer offers capabilities that are more abstract as compared to those offered by lower-level layers. Conversely, each layer is built using the elements contained in the layer above.

A realization of the MDA approach consists of Meta-Object Facility (MOF), Unified Modeling Language (UML), User Model, and User Object modeling layers (OMG 2003). At the meta-meta model (M3) layer, the MOF has an Ecore specification for defining metamodels in the OMG's family of MDA languages. Described using the UML metamodel, the M3 layer supports computation-independent metadata management, metadata services, model management, tag capability, and reflective operations, among others. The metamodel (M2) layer can have models that conform to the M3 layer. The M2 layer is for platform-independent modeling. These models can be domain-specific. The Ecore at the M2 layer can be used to define concrete models at the M1 layer. The M0 layer is used to describe instances of models specified at the M1 layer. The M3, M2, M1, and M0 layers support the incremental development of models for component-based systems. It is useful to note that the separation of concerns in MDA is essential for developing software system tools, including simulators.

2.1.2 DEVS Atomic Model

The set-theoretic specification of parallel atomic model $X, S, Y, ext, int, conf, , ta$ is domain-neutral. Its input and output are defined in terms of port names and variables. The variables can be arbitrarily complex. And atomic models are responsible for handling differences in the input and output variables. From software design, appropriate I/O type consistency is necessary. For any user-defined (and domain-specific) model, the internal, external, and confluent, time advance, and output functions can have arbitrary logic as long as they satisfy the abstract definitions provided in the mathematical atomic model specification. A restricted specification of parallel DEVS called Finite Deterministic DEVS (FD-DEVS) introduced by Hwang and Zeigler (2009) has been developed. Events and states are defined to be finite sets, and external and internal events are allowed to occur at time intervals restricted to rational numbers.

No time interval between one event and the next can be infinitely small. Abstracting time to be rational instead of real numbers is one way to achieve that. When states are simple, possible state transitions can be enumerated and unreachable states identified. These restrictions can simplify model validation for the EMF-DEVS modeling described next.

2.1.3 EMF-DEVS Atomic Model

Sarjoughian and Markid (2012) propose EMF-DEVS as a metamodeling approach for the parallel DEVS formalism. The basic aim is to define and validate DEVS metamodels using the Eclipse EMF framework. The EMF validation infrastructure is used to define the elements of DEVS models with a set of constraints. These constraints align with the DEVS formalism and the target DEVS-Suite simulator, which uses the Java programming language. The structures of atomic and coupled meta-DEVS models can be modeled and validated. The generic capabilities provided in the EMF M3 and M2 layers are extended to support concrete models for the DEVS-Suite simulator. The EMF-DEVS metamodel can support input, output, and state sets as well as external, internal, output, and time advance functions. These abstract functions $(\delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ do not include the logic that is necessary to define behaviors. For example, the external transition function δ_{ext} does not define a generic transition from a source state to a target state with constraints. Also, the output function does not define conditions for generating outputs.

In the context of metamodeling as in EMF-DEVS, the term validation refers to the Eclipse EMF validation framework and its execution engine. The Eclipse EMF has built-in validation mechanisms such as reflection for the metamodels at the M2 layer. Metamodels at the M2 layer can be validated for conformance to the meta-meta model at the M3 layer. Concrete models at the M1 layer can also be validated to

conform to the DEVS metamodel. Here validation does not refer to the execution of a metamodel over some time and determine whether or not it produces behavior per user requirements and expectation. Given a concrete simulation model (M1 layer), the model can be verified to be specified correctly both in terms of M1 and M2 layers. When executed over some time while recognizing the behavior as acceptable for some defined experimental condition, the model is said to be valid. Regarding the verification and validation definitions for concrete models, the EMF-DEVS validation may be referred to as verification when a metamodel has domain knowledge. For example, the external transition function has the necessary control structure and other details to specify the next state of a model given its current state and received input.

2.2 Foundational UML Subset

We examine the state of the art of UML standards and the recent advancement of the fUML and concepts like executable modeling. Therefore, it would be useful to provide an essential but yet sufficient background of the fUML and some of its primary subjects.

2.2.1 *UML Activities*

The fUML subset is devised based on the approach of activity modeling. From a high level, an activity can be seen as a directed graph of vertices and connected by edges. Each vertex is an activity node, and each edge is an activity edge which can be either an object or a control flow. Every control, object, or executable node is an activity node. Action is an executable node that can be further specialized to encompass a variety of different basic behaviors. Previously, we discussed with details the use of activities in specifying the behavior of atomic DEVS models (Alshareef

et al., 2016). Lifting behavior to higher levels across the meta-layers such as M2 can result in greater benefits, especially for checking the syntax and semantics of the specified models at a higher level of abstractions. As a result, modeling at the lower levels becomes simpler and less detailed since the other details about the system under development have been addressed at the higher levels. Also, we extend our research toward a direction on simulating UML activities by exploiting the capability provided in the Parallel DEVS simulator.

Parallel DEVS was proposed by Chow (1996) to provide the capability of handling collisions that may arise during the interaction between different components. The formalism allows all imminent messages to be sent out simultaneously, which can be used as a useful abstraction for handling activities flows. It is especially important in the case where the activity node has multiple incoming flows, such as the case in the fork and merge nodes.

2.2.2 *Abstract Syntax*

The abstract syntax of fUML mainly consists of classes, common behaviors, activities, and actions. It selects some certain elements from the complete UML to provide a more precise definition of their semantics. Although it uses the above four packages, it excludes some features thereof. Some packages from the UML 2 Superstructure are excluded. The reasons for excluding some packages or features vary. Some packages are excluded due to their insignificance in terms of execution. Others are excluded due to the encountered complexity if they are to be realized in a computational platform. On some occasions, features are excluded because of their generality. Therefore some ambiguity or restrictions concerning their semantics may arise.

The packages for activities and actions are covered, although with exclusions to some of their features. Overall, fUML does not entirely realize the complete model

of activities as defined in the UML but instead recognizes an essential subset of it. The included pieces of these packages are organized and sub-organized in a well-established architecture. The relationships are then established as necessary, using dependency, specialization, and import. Actions are the fundamental units of the behavioral specification. However, they have to be contained in some behavior that is currently the activity. According to UML 2.5, the action is a subtype of the executable node, which is itself a subtype of activity node. Therefore, actions can be contained in activities as executable nodes or interactions as actions. Currently, we only focus on their containment within activities. Action may have input pins, output pins, or constraints. Action is also specialized further to represent many different forms of behavior based on different semantics and usage. For example, structural feature actions are used to handle reading, writing, adding or removing structural features such as a queue within some model. The manipulation of the feature has to go through this set of actions. There are multiple sets within the metamodel of actions. Each set also has more subsets or specific actions. The relationship between actions is determined mainly via generalizations from the more general or abstract action to the more specific or concrete ones. In our work, we handle the concept of action in general. However, we also detail some defined actions that are necessary for the discussion and demonstration. Other sets of actions can benefit from a similar approach.

2.2.3 *Execution Model*

The execution model is a major contribution of fUML given that its abstract syntax is already defined in UML superstructure and yet revisited in the subset. The execution model is a fUML model that defines the specification for the execution. The execution model expects a well-formed model to provide a meaningful execution semantics. The operational specification is currently described in the form of equiva-

lent code in Java. The reason is that activity diagrams become inconveniently large when addressing relatively significant behavior. One of our concerns in this work is to overcome this issue by utilizing DEVS capabilities for addressing behavioral specifications.

2.2.4 Foundational Model Library and Base Semantics

It can also be useful to provide a brief description and reasoning about these two components of the fUML specifications. The foundational model library includes primitive types and behaviors accompanied by their basic operations such as functions for handling Boolean signatures. It also defines some capabilities for managing input and output. These definitions, along with other concepts, are rigorously defined in the DEVS formalism as well as the modeling and simulation packages provided in the DEVS-Suite simulator (ACIMS, 2017b). Therefore, we focus our use on the formalism and what is in the selected simulator. Regarding the base operational semantics of fUML and Parallel DEVS, the simulator provides an explicit protocol for the behavior of the execution model, which can be used for verification purposes. In our case, the semantics are substantially extended due to the definition of time. However, the base semantics are still maintained.

2.3 Formalisms and Languages for Discrete Event Modeling

There exist formalisms, modeling languages, and frameworks to develop behavioral models. Our work focuses on the rigorous specification of DEVS as an abstract mathematical formalism accompanied by a framework supported by modeling languages and run-time execution. In the following sub-sections, we describe necessary background details for understanding and developing behavioral models.

2.3.1 Parallel DEVS Atomic Model

The set-theoretic specification of the atomic model is an abstract representation of a standalone component of a system (Zeigler *et al.*, 2000). The formal specification can be defined independent of any language and, more generally, simulation platforms. From a software standpoint, we need to have the specifications to be formulated in terms of modeling and software programming languages. Many DEVS simulators can accept the specification of a model following a target simulator’s programming language syntax, semantics, and specialized constructs such as model initialization. Examples of these tools are DEVS-Suite (Kim *et al.*, 2009) and CoSMoS (Component-based System Modeling and Simulation) (ACIMS, 2017a) where the programming language is Java. Other simulators use different languages as an input such as CD++ (Chidisiuc and Wainer, 2007) and PowerDEVS (Bergero and Kofman, 2011) where the programming language is C++. The work by Hollmann *et al.* (2015) provides a specific language based on the formal specification definition language with a set of rules to translate it into simulatable models targeting simulators like DEVS-Suite and PowerDEVS. As defined by Zeigler *et al.* (2000), the basic formalism of parallel DEVS model is an algebraic structure – atomic model = $\langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$. X is the set of input events. S is state representing the tuple of sequential states. The state must have at least two independent variables. One is called *sigma* (σ), the time duration allocated to the current state of the model. The other variable, called *phase*, represents a set of state values that change and can be tracked. Y is the set of output events. δ_{int} and δ_{ext} are the internal and external transition functions, respectively. The model receives a bag of inputs meaning that the elements of the bag may have multiple occurrences and have no order. The receiving model accounts for this possibility to perform proper handling of the inputs. δ_{con} is the confluent

transition function, which can be specified to handle the collision between external and internal events. λ is the output function which transforms S into Y at arbitrary time instances. ta is the time advance function which maps the internal state into a positive real number using elapsed time since the last state transition (i.e., it computes σ which can range from zero to infinity, inclusive). Any domain-specific definition of the functions mentioned above must satisfy their corresponding abstract definitions as provided in the modeling formalism. Together, the elements of the DEVS specification allow the modeler to define operations and controls for system structure and behavior flexibly.

Simple processor

This example is selected to demonstrate some concepts throughout the discussion. We start with a simple processor and later extend to have a queue to demonstrate behavioral expressiveness. The simple processor only stores jobs upon their arrival, process them for some amount of time (duration), and then sends them through the output port. It does not account for input buffering and preemption of a job under processing. We devise the behavioral specification of this process, as described in the DEVS formalism, in a set of activity models as an intermediary phase between them and their concrete manifestations. The simple processor example as presented by Zeigler and Sarjoughian (2003) is defined as

$$Processor_{processing_time} = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle,$$

where

$$IPorts = \{“in”\}, \text{ where } X_{in} = J \text{ (a set of job identifiers);}$$

$$X_M = \{(p, v) | p \in IPorts, v \in X_{in}\} \text{ is the set of input ports and values;}$$

$$S = \{“passive”, “busy”\} \times \mathbb{R}_0^{+\infty} \times J;$$

$$\begin{aligned}
OPorts &= \{\text{"out"}\}, \text{ where } Y_{out} = J; \\
Y_M &= \{(p, v) | p \in OPorts, v \in Y_{out}\} \text{ is the set of output ports and values;} \\
\delta_{int}(phase, \sigma, x) &= (\text{"passive"}, \infty, x); \\
\delta_{ext}((phase, \sigma, x), e, ((\text{"in"}, j_1), (\text{"in"}, j_2), \dots, (\text{"in"}, j_n))), & \quad j_i \in J_{in}, \\
&= ((\text{"busy"}, processing_time), j_1, j_2, \dots, j_n) \quad \text{if phase} = \text{"passive"} \\
&= ((phase, \sigma - e), x) \quad \text{otherwise;} \\
\delta_{con}((s, ta(s)), x) &= \delta_{ext}(\delta_{int}(s), 0, x); \\
\lambda(\text{"busy"}, \sigma, j) &= j; \\
ta(phase, \sigma, j) &= \sigma.
\end{aligned}$$

2.3.2 Statecharts

There are many modeling languages available for specifying the behaviors of atomic DEVS models. Statecharts is popular due to its expressiveness power for representing complex behaviors of systems (Harel and Politi, 1998). Statecharts define mainly hierarchical states and state transitions. They can be used to specify the discrete behavior of a system and its components. Other languages and metamodels also exist for modeling the behavior such as behavior diagrams in UML (OMG, 2012). They provide different notations where behavior can be captured in various diagrams. Each diagram generally has some advantages relative to some other diagrams. The diagrams vary in their syntax as well as semantics to satisfy different needs. One major diagram is UML state machines, which is considered to be a variant of Harel's statecharts. The transitions, as well as states, can be associated with some behavior. Modelers can use state machines, interactions, sequences, or activities for describing behaviors within and across model components.

2.3.3 Activities and Actions

The UML Activities diagram is a major method for developing detailed behavioral models (OMG, 2012). Their standardized specifications, including visual syntax and semantics, have undergone significant changes under the stewardship of the International OMG Standardization consortium. Activities allow for behaviors to be specified using a set of elements along with their sequencing defined as control and object flows. The elements are the activity nodes. A node can be control, object, or executable node. Control nodes define the flow of tokens between activity nodes. A control node can be either *initial* to define the starting point of an activity execution, *final* to define when activity stops, *fork* for splitting a flow into multiple flows, *join* for synchronizing multiple flows, *merge* to act similar to join but without synchronizing flows, or *decision* to select between its outgoing flows. Each control node can be used to define certain behavioral properties of components such as the DEVS atomic model. Object nodes are used to handle data. We will use the activity parameter node to define inputs. Finally, executable nodes are the core elements of activities. Actions are a special type of executable nodes, and therefore, they can be within an activity. On the other hand, actions in UML 2.5 (OMG, 2012) can be only defined in the context of an activity. Thus, together, they provide a means for modeling behavior to establish processing routines that include control structures. It is important to note that activity modeling emphasizes and supports specifying actions and combining them using arbitrary control structures.

Actions are the only kind of the executable nodes in UML 2.5. They are necessary to take advantage of more capabilities provided by the activities. Besides, they are the fundamental units of behavior specification. Some actions change the state of the system. This kind of action in our approach satisfies how states are changed in

DEVS atomic model internal and external transition functions. Examples of these actions are *add structural feature value* and *value specification* actions. Other kinds of action support handling objects. *Read self* action is used to obtain the current object context and place it in its output pin. *Value specification* action is also used to provide certain value and place it in its output pin. The structural feature actions are used for either assigning or retrieving a structural feature of an object. They can be both used for the *phase* as a structural feature. They can also be used for other features or more complex objects.

Event actions can also be used. For example, an *accept event* action can be used to model the waiting for an event to occur in some other entity to proceed in the activity flow. In the context of UML actions, this event can be caused by the simulation protocol to trigger some atomic model components, or by other models that decompose the behavior into multiple models. There can also be an *accept time event* action which is used to model waiting time. *Send signal* action can be used to model invocation for some other components. It is also used to enforce some order when used with the accept event explicitly. For example, to impose the order of executing the DEVS output and internal transition functions, a sending event signal must be completed to enable the accept event action. However, this sort of scenario is part of the simulation protocol.

Additionally, invocation actions provide a means for communication and signals among multiple activities, e.g., *call behavior* action, to call either behavior or operation. The action can be synchronous if it has to wait for the called behavior or operation to complete. Otherwise, it can be asynchronous and then immediately proceed after calling the associated behavior or operation. Such an order is crucial for the execution of the behavior of the atomic model. That is, internal and external transition functions cannot execute simultaneously although possible, using the

confluent transition function. We will elaborate further in the subsequent sections.

2.4 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) (Steinberg *et al.*, 2008) is the core of the Eclipse modeling project, which serves as a basis for modeling and metamodeling. The framework and its various capabilities provide an Eclipse realized manifestation of the Model-Driven Architecture (MDA). It is surrounded and equipped with a variety of tools to facilitate the process of creating and maintaining metamodels as well as producing sets of classes and runtime support in highly model-driven development. The Ecore as the basis for EMF is used to define metamodels to provide a common grounding for UML, XMI, and Java. It unifies these and other models in a well-defined setting with automated mapping from one to another (e.g., UML to Java). Such incorporation places a strong focus on the model as the fundamental unit for building software-based systems.

EMF has been extended to provide a means for creating metamodels for parallel DEVS formalism as in EMF-DEVS Sarjoughian and Markid (2012). The framework currently supports defining and validating the structure of atomic and coupled meta-DEVS models. As discussed earlier, we want to take a step forward into behavioral modeling, which turned out to be not straight forward. The EMF itself is mainly concerned with structural aspects. However, in a previous work (Sarjoughian *et al.*, 2015), we discuss behavioral metamodeling for DEVS by extending the Ecore. In this work, we realize the activity metamodel as a significant step in creating a behavioral modeling engine and preparing for further support, such as defining the graphical definition and mapping.

Graphical Modeling Framework (GMF) (Gronback, 2009) has been built upon EMF. It exploits EMF capabilities such as code generation and model serialization

to specify models visually. These models are ensured to conform to their metamodels defined in Ecore. GMF, as an extension of EMF, allows for building tooling infrastructure to be used for modeling and diagram generation. This approach is used in developing an engine enabling specification of statecharts for DEVS atomic models (Fard and Sarjoughian, 2015). Although this approach uses both EMF and the Graphical Editing Framework (GEF), it segregates the metamodel from the graphical information in a clear manner. This is especially useful in our case since we need to support graphical notations for activities as well as maintaining consistency between the structural and behavioral metamodels for DEVS modeling. The metamodel is referred to as a domain model in the context of GMF. Once defined, there are two models to be initially generated and then manipulated to account for the specific graphical requirements. Those two models are the graphical and the tooling definition model. The graphical definition model defines graphical components and figures used in the models. The tooling definition model defines the other aspects of the editor, such as the palette. The generation model is generated similarly to EMF. After that, the mapping model combines all definitions in the three previous models to put things together and map every element to its graphical counterparts. Once created, the process of generating the diagram editor becomes ready to be performed.

BEHAVIORAL DEVS METAMODELING

A variety of methods may be used to represent time-based dynamics of systems. The behavior of a system, for example, can be modeled using set-theory, UML diagrams, and pseudo code. Each kind of model serves specific purposes and must ultimately map to programming code suitable for execution in one or possibly multiple target simulators. A mathematical model is useful for defining a systems structure and behavior independent of software design and simulation technologies. UML Class and Statecharts diagrams, among others, help design complex modeling and simulate engines that may or may not necessarily have mathematical grounding. Computer code can be developed and partially generated based on mathematical or certain kinds of software specifications. Each of these methods has its strengths and weaknesses, and none is currently considered to contain all the necessary capabilities required for generating executable simulation code.

The atomic and coupled models in the DEVS formalism (Zeigler *et al.*, 1997) are metamodels. From the standpoint of MDA, DEVS has an abstract syntax and an execution semantics that together define a modeling language for discrete event systems. The set-theoretic DEVS models are abstract mathematical artifacts. An atomic DEVS has its elements defined, for example, as sets, functions, and relations. These model elements individually and collectively satisfy certain general abstract properties and constraints. For example, a model can receive a finite number of input events within a finite period of time at arbitrary time instances, process these inputs with state changes within a period, and generate a finite number of output events. It is the responsibility of the modeler to show that the developed atomic models for a

given target simulator satisfies the properties and conform to the constraints defined for the DEVS atomic formal specification.

In the MDA framework, a concrete atomic DEVS model for a system component, relative to its metamodel, has specific structural (e.g., inputs and states with possible particular values) and behavioral elements (e.g., state transitions for a particular source and target states with assigned times to next events). The metamodel is a language within which concrete models can be developed. Furthermore, a concrete model may also satisfy constraints such as state variable types and state transitions sanctioned for specific application domains. Full-fledge behavioral DEVS metamodeling can support the automatic conformance of concrete models to their metamodels. This capability can significantly reduce the amount of manual effort required to show concrete models that satisfy their metamodel properties and constraints.

From a tool's perspective, a simulator, such as DEVS-Suite (ACIMS, 2017b), is designed as a collection of UML classifiers and relations that capture some aspects of the set-theoretic atomic and coupled parallel DEVS models. These models can also be collectively referred to as a DEVS UML metamodel. The inputs, states, and outputs, and internal, external, output, and time advance functions of the model are defined abstractly; they, by themselves, are not executable. For example, the data structure for input is defined as a pair (port-name and input-variable) where the port has a string type, and the input variable has an entity type. Similarly, the external transition function is defined as a method with specific arguments, but without any actual implementations for the state transitions and conditions under which they are to execute. As in its mathematical counterpart, a concrete atomic model must have instances of the port-name and input-variable attributes belonging to the UML classes and interfaces. The realization of the formal DEVS models as UML specifications is advantageous. UML includes abstractions such as data typing, return types, and

control structures that enrich the abstract atomic DEVS model specification. These models can map through transformation into partial code for programming languages using professional tools dating back to the 1990s.

Simulators such as DEVS-Suite do not explicitly account for domain-specific modeling. A modeler can develop domain-specific models using object-oriented modeling principles and design patterns. Low-level techniques can enforce in an ad-hoc manner the domain-neutral contracts embodied in the DEVS UML models. Examples of such techniques are checking for data type compatibility and expected values for concrete models. Eventually, these models are to implement in some specific programming languages. These contracts cannot account for domain-specific knowledge; they must encounter extensions. This approach becomes complicated and unwieldy as the scale and complexity of the system, that is to be simulated, increase. Such resulting simulators lack rich capabilities to support and develop domain-specific metamodels and also are unable to validate basic model properties and constraints such as data typing and legitimate state transitions, for example. MDA-based modeling, however, can lend itself to develop and automatically validate the behavior of any domain-specific DEVS concrete model against its metamodel and by extension the general-purpose atomic DEVS model.

Given the above discussions, we can make a few observations. When concrete atomic DEVS models are developed using programming languages, it is challenging to ensure they conform to their abstract model. A substantial amount of effort is required to concretize behavioral abstractions. Therefore, it is essential for the meta and concrete atomic models to be systematically related to each other as proposed in the MDA framework. This relation is especially important, given that the challenging part of developing models of complex systems is specifying their behaviors. Therefore, we need an atomic DEVS metamodel that can support behavioral modeling (e.g., re-

ceiving sanctioned input events and legitimate state transitions with timing). Toward this goal, we propose behavioral metamodeling for the general-purpose and domain-specific atomic models using the Eclipse Modeling Framework (Steinberg *et al.*, 2008). Consistency between these models can be specified and enforced (referred to as validated) with automation. Concrete models can generate from their domain-specific metamodels. Behaviors contained in these metamodels can significantly reduce the amount of effort to create concrete models and improve their quality using automated code generation.

3.1 Related Work

In this section, we primarily focus on behavioral DEVS atomic metamodeling and briefly consider the extent to which detailed specifications can be supported. Model-driven design approaches have been playing a more significant role in developing complex simulation models. Focusing our attention on the OMG MDA framework and DEVS, we find some approaches that follow the MOF Technology Space (Bézivin *et al.*, 2005). Yonglin *et al.* (2009) proposed a DEVS metamodel for developing SMP2 (Simulation Model Portability standard). This metamodel maps to SMP2 metamodel using QVT (Miller and Mukerji, 2003). Simple states and state transitions for the atomic DEVS model are supported. In work by Cetinkaya *et al.* (2012), structural DEVS metamodeling can be supported. As in EMF-DEVS, behavior specification for the atomic DEVS metamodel is not supported (see Section 2.3).

In the MOF technology space, some works have employed DEVS Natural Language (DNL), XML Schema, and Extended BNF for defining DEVS models. These support behavioral modeling using mostly the same ideas and methods. The MS4Me (Seo *et al.*, 2013) focuses on modeling using DNL as described by (Zeigler and Sarjoughian, 2012). The DNL as meta-language supports Finite-Deterministic DEVS

models (Hwang and Zeigler, 2009). MS4Me uses Xtext (Eclipse Foundation, 2013) to enforce DNL rules for simple inputs, outputs, states, state transitions, and timing. As a modern Java-like language, Xtend supports developing FD-DEVS models. The MS4Me models can be augmented to become Parallel DEVS models using the full expressiveness of the Java language. It supports adding Java code to the model and thus developing Parallel DEVS models while maintaining a tight connection with the FD-DEVS models. The Java code is injected into slots in a structured manner using tagged code blocks. These are inserted directly into the generated source files. These tagged code blocks are used to specify additional behavior for initializing, internal transition, external transition, and output. Compared with FD-DEVS, classic, or parallel DEVS models that have these kinds of code blocks are difficult to validate. The DEVSML (Mittal *et al.*, 2007) is developed for DEVS simulation models that can be executed in net-centric computing environments.

Some works employ SysML (Nikolaidou *et al.*, 2008) and UML (Borland *et al.*, 2003; Risco-Martín *et al.*, 2009; Mooney and Sarjoughian, 2009; Pasqua *et al.*, 2012). The authors developed a SysML profile for classical DEVS. An atomic model is defined as a collection of stereotype blocks. State Definition and Association diagrams define the behavior. Atomic Internal and External diagrams describe the internal and external functions, respectively. Descriptions for the time advance and output functions are parts of the Atomic internal diagram. Similar to the above approaches, simple states with constraints are defined. The external diagram follows FSM with control elements such as choice, fork, and join elements. Time allocated to states can only be defined in the internal diagram. The DEVS SysML profile and DEVS MOF are intrinsically different due to their technology spaces. There exist other approaches that use metamodeling abstraction (Fard and Sarjoughian, 2015; Ighoroje *et al.*, 2012; de Lara and Vangheluwe, 2004). Garredu *et al.* (2014) give a survey that discusses

the uses of some MDE approaches for DEVS.

3.2 Atomic DEVS Metamodeling

The mathematical properties and constraints defining an atomic DEVS model can apply to any implementation of it. Therefore, it is useful to have a framework that can not only capture the atomic model's formal specification (i.e., a metamodel) but also enforce its syntax and semantics for domain-specific metamodels. Another important advantage is to define models independent of any particular simulator. That is, metamodels can transform into concrete models that can execute in simulators implemented in specific computing platforms. This framework must (help) validate the behavior of any concrete atomic DEVS model against its metamodel. To achieve this, we propose introducing behavioral metamodeling to structural metamodeling. The resulting metamodeling framework must also lend itself to developing metamodels for modelers domains of interest. This framework is also desired to support defining domain-specific concrete models for desired systems.

We intuitively define behavioral metamodeling as a set of concepts realized in a framework that supports specifying operational details of the internal, external, output, and time advance functions of any atomic DEVS model. These generic operations can be used to define behavior for any domain-specific DEVS metamodel. Domain-specific behavior can be specified by extending the generic DEVS metamodel behavior. That is, the behavior of these functions is defined independently of computing platforms in which they can be fully implemented. The properties and constraints in the domain-neutral and domain-specific functions for the concrete models can be validated. The properties and constraints of the functions that are not satisfied in any concrete model are automatically identified and reported.

Figure 3.1 illustrates the concept of meta and concrete mathematical and UML

modeling. The structure, unlike behavior, of mathematical atomic and coupled DEVS models can be completely specified both abstractly (as a metamodel) and concretely (as a concrete model). In mathematical modeling, a concrete model has more information relative to its metamodel. In the metamodel, δ_{ext} , δ_{int} , δ_{con} , λ , ta functions are abstract mathematical constructs. The abstract atomic DEVS model functions do not have sufficient details, for example, as in Statecharts. Indeed Statecharts also does not capture the levels of detail in the functions that an arbitrary atomic model can have. In contrast, arbitrary concrete atomic models must have details, including decision logic and control in the state, output, and timing functions.

The concept of meta and concrete models in UML are distinct as compared with the ones just described for a mathematical model. While UML metamodels are independent of computing platforms, concrete-models are not. Separating models to be platform-independent and platform-specific is important (see Section 2.1). Metamodels are technology (simulator) agnostic. Concrete models include details that are specific to target simulators. The meta and concrete models can be related to one another.

Focusing on behavioral modeling, the line arrows from the concrete model and metamodel are conceptual. For mathematical modeling, one may construct relationships to show, for example, state transitions in an external transition function in a concrete model conform to the abstract external transition function specification. In UML modeling, one can include rules that can apply to concrete models. The block arrows at the metamodel and concrete model levels involve complex modeling and software development tasks, requiring detailed design and code development.

Considering the distinct roles mathematical and UML modeling offers, a desirable goal is to support both. The EMF framework (Steinberg *et al.*, 2008) is a strong candidate as it already supports UML meta- and concrete modeling, and it can sup-

port developing specific metamodels as in EMF-DEVS. In particular, the relationship between meta (M2 layer) and concrete models (M1 layer) is formalized. Furthermore, the EMF includes the meta-meta model (M3 layer) and instance models (M0 layer). Given these, we extend the EMF-DEVS (Sarjoughian and Markid, 2012) structural metamodeling to enable behavioral (functional) metamodeling. Generic and domain-specific metamodels with built-in and user-defined properties and constraints for the external, internal, output, and time advance functions are supported. Modelers may develop metamodels in a structured setting, thus leading to the automation of meta-model validation as defined in EMF. (We note that validation is not referring to simulation validation.) Constraints defined for the generic and domain-specific atomic DEVS metamodels enable validating concrete atomic models.

3.2.1 Meta-Behavior Modeling in EMF

We begin by sketching the basic details of the M2, M1, and M0 layers for the atomic DEVS model shown in Figure 3.1. At the M2 layer, the Ecore is an instance of the Ecore at the M3 layer. The M3 Ecore metamodel is at a higher level of abstraction for the atomic DEVS metamodel. That is, the DEVS metamodel extends the instance of the M3 Ecore. The role of the M2 layer is to support developing concrete models at the M1 layer.

As noted earlier, the DEVS-Suite simulator is developed in Java, a strongly typed language. The kernel of the modeling engine contains data structures and operations that satisfy the DEVS modeling formalism. Thus, at the M1 layer, user-defined models can be generated from the DEVS metamodels. Suppose we want a Processor model which can receive bags of input, process one of them, and generate one or more outputs. Assuming we have an eProcessor metamodel, it can be used to create the concrete Processor model. This concrete model at the M1 layer can be created for

a platform-specific simulator, such as DEVS-Suite. The DEVS-Suite simulator can execute an instance of the concrete model at the M0 layer.

In MDA, the M0 layer refers to the instances of the user models. These can be physical objects or executable software objects (e.g., compiled code). Such instances can be modeled as UML Object diagrams. As software objects, they can exist at execution time, and their states may be stored, for example, like XML or byte code. In contrast, for simulation, the M0 layer refers to the users parameterized atomic and coupled models. Therefore, at this layer, we have not only parameterized models but also their instances as part of other coupled model instances (see Figure 3.1).

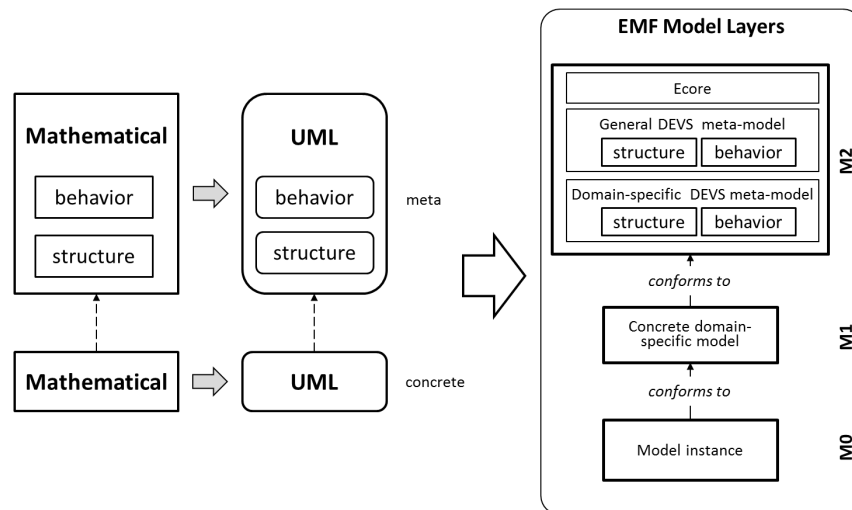


Figure 3.1: From Mathematical to UML to EMF Modeling.

Although metamodeling is not as expressive as programming languages such as Java, it is shown to be useful, for example, as in the Graphical Modeling Framework (Gronback, 2009). The metamodel behavior specification for DEVS functions is achievable using Statecharts (Harel, 1987). The elements of a parallel atomic model at M1 can be arbitrarily complex. An example is the external transition function. It can have any attribute type, expressions, and control structures that a target computing

platform supports.

The signature definitions for the atomic model external and internal transition functions can be defined using structural metamodel as in EMF-DEVS. The abstract definitions for these two functions must include some operations needed to result in some appropriate state change. State changes in these functions can be defined as transitions amongst source and target states. A transition may have input events, conditions, and actions. A prototypical state transition is set to transition from a source state to a target state. Such a constraint for state transitions can be identified and validated at the M2 layer. The output and time advance functions can also be set using operations and control structures. An operation can have attributes and statements (McNeill, 2008). A metamodel behavior specification requires identifying abstractions for state transitions in the external, internal, and confluent transition functions.

Similarly, appropriate abstractions are needed for the output and time advance functions at the M2 layer. The behavior of all DEVS functions as just described, can be validated using EMF. The definitions for the atomic model functions must be consistent with the abstract DEVS simulation protocol.

To model the content of EOperation, we need to extend the EMF Ecore metamodel (McNeill, 2008). Therefore, we will extend the Ecore metamodel to model DEVS functions that have been defined as EOperations (i.e., interface definitions) in EMF-DEVS. Our goal is not just to validate domain metamodels. We also aim to execute these functions after concrete models are generated for a specific simulator, DEVS-Suite, for instance. The code generation creates the corresponding code for the defined elements in the metamodel. In EMF, the generator model plays a significant role in how the resulting code could be generated and organized via some settings that may differ based on the targeted platform. Those settings can be configured separately

to ensure that the model maintains its platform independency. The process can be manipulated in a way that will lead to producing concrete models.

Thus, the general metamodel, shown in Figure 3.2, extends the EMF Ecore metamodel with some definitions for state transitions, actions, and conditions. It also includes essential elements of the atomic DEVS model. The metamodel extends Ecore elements with DEVS functions and also others for defining behavior. By extending Ecore, we are enabling EOperation (which is used to define DEVS functions) to include some content that can transform into the concrete code rather than just having operation signatures. The extended EOperations will be contained in the extended EClass (eAtomic in our case) since they cannot be contained in EClass itself. This is a reason for extending EClass and EPackage since the Ecore elements themselves (EClass and EPackage) will not allow adding the extended ones (Extended EClass and EOperation) McNeill (2008). Therefore, we first extend EOperation as a basic step to support behavioral DEVS metamodeling. Second, we extend EClass to allow adding the extended EOperation. The third step is extending EPackage to enable adding the extended EClass.

The second part of the metamodel (shown in the middle of Figure 3.2) is specializing eDEVSOperation to represent external transition, internal transition, output, and time advance functions. All of these can include operations that have statements and local variables. They also may have return values. The eDeltExt and the eDeltInt represent external transition and internal transition functions. Both compose transitions defined to capture the concept of state transition. State transition has a name defined as an EString, source, and target defined as an ETypedElement, input defined as an optional reference of type eInput to be used in the external transition function. It can also have some actions and conditions. We also added two specialized state transitions for the phase and sigma primary states. Source and target

phases are added to the state phase transition (StatePhaseTransition) and defined as an EString. Source and target states for sigma are added to the state sigma transition (StateSigmaTransition) and defined as an EDouble. Any other specific state transition can also be defined in the same manner for domain-specific models.

The behavior is consistently captured at the general and domain-specific meta-modeling at the M2 layer. The generic behavioral metamodel is predefined for the modeler. The domain-specific meta-behavior can be defined by the modeler as needed. The same approach is followed for the actions and conditions that are represented abstractly and then specialized in providing the support for developing the behavior at the concrete model. The eOutput and eTA elements refer to the eState in addition to the inherited composition feature from eDEVSOOperation to support having other operations for more functionalities.

3.2.2 Constrained Meta-behavior Modeling

The metamodel shown in Figure 3.2 is based on the parallel atomic DEVS model. This model has an infinite state-space, and therefore model validation (as in model checking) is impractical. A sub-class of DEVS called Finite-Deterministic DEVS (FD-DEVS) (Hwang and Zeigler, 2009) has finite state-space, which makes it attractive for behavior modeling at the M2 layer. The total state of the atomic DEVS metamodel can be defined as $\{primary\} \times \{secondary\} \times \mathbb{R}_{[0,\infty]}$. An atomic FD-DEVS model restricts the range of values for the time advance function to $\mathbb{Q}_{[0,\infty]}$. Model validation is computable when the values for inputs, outputs, and states (including the time to next event) are finite. These constraints can be validated for having legitimate output, time advance, and internal and external transition functions. Constraints for state transitions (belonging to both external and internal transition functions) can be validated. For example, states in any state transition can be validated to include

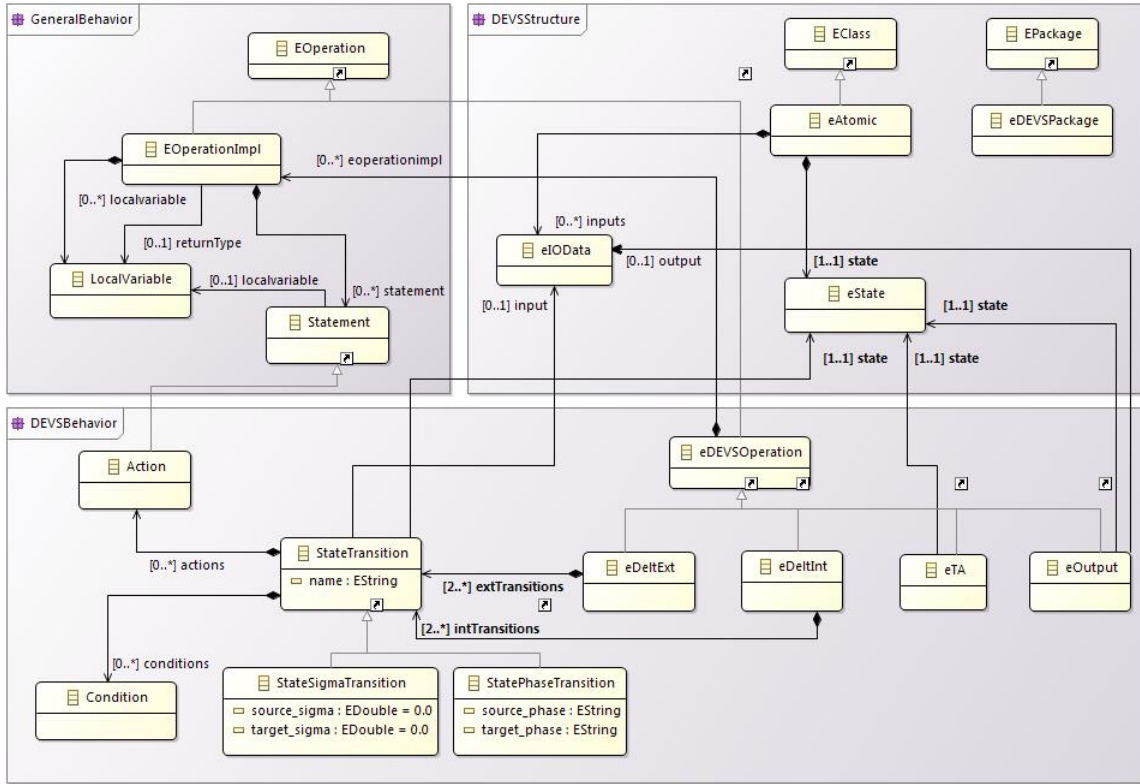


Figure 3.2: A Metamodel for Atomic DEVS Model with State Transitions.

only the states defined in the model's state set, and there are no unreachable states. For the external event, its input event can be checked to be included in the input set. State to output mappings can also be validated by checking whether or not every output belongs to the output set. We can also check if outputs are computed using states that belong to the state set. Time to next event for every state transition must also belong to $\mathbb{Q}_{[0,\infty]}$. When the time interval is infinity, there is no output. Validation of behavior domain-knowledge can be augmented with user-defined constraints.

Considering a domain-specific metamodel, they may have their constraints on the input, output, and state sets as well as the atomic model functions. These constraints must be defined by the user, for example, by extending the EMF-DEVS metamodel. Users may specify domain-specific constraints using the EMF Eclipse framework and

tool. Of course, user-defined constraints cannot contradict those that are defined for the generic metamodel. We note that the restrictions in the atomic FD-DEVS model and its dynamics may require complex control structures. State transitions in the external (or internal) transition function may have to be synthesized in complex patterns. Transitioning between external and internal transition functions can have many configurations.

Similarly, the output and time advance functions may have complex structures. These considerations restrict the behavioral metamodeling describe above. Nonetheless, the capabilities afforded by MDA is advantageous as compared with model development where there is little or no means to start from metamodeling and reach executable models. Specific state transitions can be individually validated at the M2 layer. Behavioral metamodeling developed in this research aids model validation before transforming them into an M1 model and M0 simulation. Once concrete FD-DEVS models are generated from metamodels, they can be validated using existing techniques and tools (Dill, 1989; Hwang and Zeigler, 2009).

3.3 A Processor Example Behavioral Metamodel Snippet

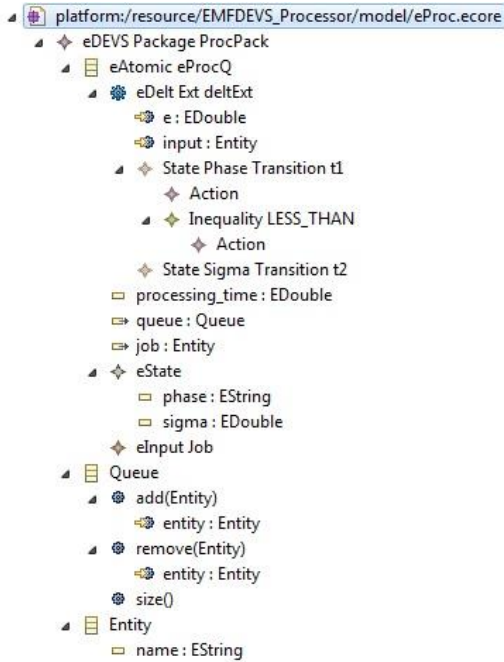
In this section, we will demonstrate the process of developing a domain-specific model (eProcQ as shown in Figure 3.3), which represents a simple processor with a queue. The processor metamodel is developed using the definition provided at the atomic DEVS metamodel. The root element is eDEVSPackage, which can contain the eAtomic models such as eProcQ and any other EClass such as Entity and Queue. Entity and Queue EClasses are defined similarly to their definition in the DEVS-Suite GenCol library (ACIMS, 2017b). Figure 3.3a shows all the model elements in the EMF editor and Figure 3.3b depicts the corresponding Class Diagram for the eProcQ Ecore model. Detailed specifications are provided for the external transition

function relative to other modeled elements such as model states and variables.

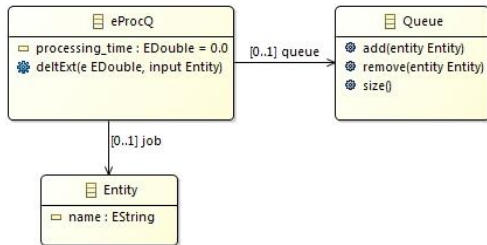
We created two transitions and gave the values associated with each one. The first transition is for the phase, and the other one is for the sigma. Figure 3.3c shows the specified properties for the state phase transition that complies with the state phase transition definition. The phase transition has a condition and an action. The condition is modeled as inequality for the queue size. The action is modeled as a method call for add operation, which is defined in Queue EClass. The action allows specifying the object, an action name that can be any operation associated with that object, and parameters. All of them have been defined as EReferences to their targeted model elements (see Figure 3.3d). Figure 3.3e shows an inequality condition specified based on the queue size. It has a left-hand side which is specified as an action (*queue.size()*) as shown in Figure 3.3f) and right-hand side which is specified as an integer value of type EInt in this case. Currently, the metamodel is limited for only those scenarios since they are the only ones defined within the atomic DEVS metamodel. The implementation is done on a Windows 7 Computer. The models are created using Eclipse Mars Milestone 6 with Eclipse Modeling Tools and EMF Ecore 2.11.

3.4 Conclusion

The term metamodel invokes different understandings since it refers to some model abstracted to another. It can encompass theories, methods, tools, and domains of discourse, including simulation. As such, metamodeling is used by theorists, developers, and practitioners in software and simulation engineering, among others. We considered the modeling formalisms, and in particular, asked at what levels of abstraction can the behavior of a prototypical atomic DEVS model be specified. Our inquiry is to distinguish meta-, concrete, and instance modeling layers from the standpoint of



(a) Ecore Editor View for the Processor.



(b) A Class Diagram for the Processor.

| Property | Value |
|--------------|----------------|
| Input | eInput Job |
| Name | t1 |
| Source | phase: EString |
| Source phase | passive |
| Target | phase: EString |
| Target phase | active |

(c) Phase Change for Transition t1.

| Property | Value |
|---------------|---------------|
| Localvariable | |
| Name | add(Entity) |
| Object | queue: Queue |
| Parameters | input: Entity |

(d) Action for Transition t1.

| Property | Value |
|----------|-----------|
| Operator | LESS_THAN |
| RHS | 10 |

(e) Less Than Inequality for Transition t1.

| Property | Value |
|---------------|--------------|
| Localvariable | |
| Name | size() |
| Object | queue: Queue |
| Parameters | |

(f) Left Hand-Side for the Less Than Inequality for Transition t1.

Figure 3.3: Ecore for a Processor with Primary State Transitions for the External Transition Function.

Model-Driven Architecture. These layers can form a basis for building a new generation of modeling and simulation frameworks and tools that can help move from

metamodeling to simulation code step-by-step. It is helpful to have modeling methods with tools that can not only represent mathematical abstractions within the MDA layers but also introduce capabilities to enforce verification and validation as much as possible in the M2 before resorting to the M1 and M0 layers.

One of the challenges facing building such ideal modeling and simulation tools is the difficulty of specifying the behavior of models. We focused our attention on the atomic DEVS model. We proposed defining meta-behavior for general and domain-specific modeling using the concept of state transition from Statecharts for external and internal transition functions (see Figure 3.3). We then extended the EMF Ecore operation with the external, internal, output, and time advance functions. These functions, unlike the mathematical counterparts, can have some of their behaviors defined. These functions can also be validated to a limited degree. To validate, we described the necessity of restricting DEVS to Finite-Deterministic DEVS. We developed an example to show behavioral metamodeling for the atomic DEVS model. We focused this work on the platform-independent metamodeling. We briefly discussed its role in developing platform-specific tools. Looking further into metamodeling, we observe that a target simulator must lend itself to the behavior defined in terms of state transitions, output, and time advance functions. Each function can have parts that are arbitrary and specific to the system being modeled. Thus, mapping behavior at a higher-level abstraction (as in the M2 layer) to lower-level abstractions (as in M1 and M0 layers) involves execution semantics (e.g., simulators may handle simultaneous event and communication differently despite being consistent with the abstract simulation protocol). Thus, it is desirable to lift behavior modeling as much as possible to the M2 layer with support to checking syntax and semantics with as little dependency as possible on the M1 and M0 layers. It is also necessary to account for simulator design/implementation choices.

Knowing the high degree of DEVS expressiveness and the MDA framework, it is easy to see approaches that such as FD-DEVS should simplify the development of verification and validation methods and tools. The degree, to which the behavioral metamodel may apply to other kinds of modeling formalisms also, remains as future work. In particular, for models that cannot be represented as DEVS, our approach for specifying meta-behavior may turn out to be useful. Finally, we believe exciting, challenging theoretical, methodological, developmental, and practical research remain to be formulated and answered for achieving general and domain-specific multi-layer behavioral modeling, including meta-modeling.

AN APPROACH FOR ACTIVITY-BASED DEVS MODEL SPECIFICATION

The Unified Modeling Language (UML) and the Model Driven Architecture (MDA) framework are commonly used to specify models of systems. They offer modeling constructs (e.g., activity node and control flow) capable of specifying DEVS atomic model behavior. Furthermore, it is possible for some UML behavioral specifications, in conjunction with the MDA framework, to be shown to conform to the DEVS formalism. Tools, supporting MDA, offer built-in capabilities to validate user-defined DEVS models in a disciplined manner. Enabling early validation of simulatable models has significant benefits, especially as systems continue to grow in complexity and scale rapidly. These benefits can be achieved once the model development environment is enriched with some formalism that has a well-defined syntax (modeling constructs) and a sound semantic (execution protocol). The DEVS formalism and its abstract simulator provide a suitable means to satisfy these needs for developing and simulating system-theoretic models (Zeigler *et al.*, 2000).

Many approaches use popular languages (such as Statecharts) for specifying the behavior of discrete-event models. Some efforts focus on restricted variants of the DEVS formalism such as FD-DEVS (Hwang and Zeigler, 2009), supporting model verification. More recently validation of DEVS behavior, grounded in UML meta-modeling and the Eclipse Modeling Framework (EMF), a realization of the MDA, has been proposed by Sarjoughian and Markid (2012). Auto-generated simulation models have also received attention with few tools supporting some basic behavior modeling.

To specify the behavior of atomic DEVS models using UML specification methods

such as Statecharts, it is necessary for the methods to conform to the DEVS formalism. It implies that both the syntax and semantics of the atomic model (structure and behavior) must be captured in UML models (such as Activity models) and their variants. Furthermore, the atomic model specification and its simulators need to be loosely coupled. That is useful for the models to be not specific to some target simulator.

The availability of software system modeling frameworks with their increasing automation capabilities is invaluable for reducing the gap between DEVS and UML abstractions (Sarjoughian *et al.*, 2015; Sarjoughian and Markid, 2012; Zeigler and Sarjoughian, 2012). Advanced architectures of frameworks such as EMF offer important functionalities across the model development lifecycle. Some capabilities are straightforward to employ the following guidelines and standards, but others require rigorous analyses and further development to apply for simulation purposes.

To define behaviors of any arbitrary atomic DEVS models, we customize the UML activities according to the atomic DEVS formal specification. We will describe the metamodel of activities in the context of atomic modeling. Due to the nature of the Activity modeling language as a subset of UML, the usage of the activity metamodel significantly varies with different views and aspects in the modeling process. Therefore, we consider different views that can be taken when employing activities to describe the behaviors of DEVS atomic models. The activity explicitly defines a set of modeling capabilities such as sequencing which leads to many possibilities corresponding to different ordering and partitioning of the behavior being modeled.

In this work, we briefly give a background about some candidate languages for developing behavioral atomic DEVS models with an emphasis on the UML activities as defined in the UML 2.5 specification (OMG, 2012). After the related work, we present our approach by going through different views for specifying the behavior

followed by presenting an activity specification for the atomic DEVS model. Then, we illustrate the usage of the actions in the atomic model, followed by an example. After that, we discuss further the relationship with the DEVS Statecharts (Fard and Sarjoughian, 2015). Finally, we conclude and briefly discuss ongoing future research.

4.1 Related Work

There are many works on the concepts, methods, tools, and technologies to make DEVS more accessible to users. We focus on some of the ongoing works that use certain UML behavioral diagrams for DEVS model specifications. Some works address the need to create models that are ready for simulation via transformation and translation techniques. They take into account some conventional approaches for modeling the behavior of systems such as Statecharts (state-based approach) and activities (flow and event-based approach). They target specific simulators where high-level model specifications can be partially translated to code that can be executed using a specific simulator. Other works consider model-driven techniques in addressing the problem of specifying behavior at different levels of abstractions. They consider metamodeling and model-driven architecture as a means for defining modeling languages that take into account the behavioral specifications of the system. Formal verification is being addressed as well in some of these approaches and frameworks using a variety of software engineering methods and tools. Our work focuses on behavioral atomic DEVS model specification. We neither focus on transforming models nor translating models to simulatable code. We aim to enrich developing the behavioral model specification.

There have been some studies in using Statecharts for modeling atomic DEVS behavior. In an early work, the use of Statecharts for defining the behavior of DEVS models was proposed by Schulz *et al.* (2000). A mapping from DEVS models to UML Statecharts is offered to support graphical DEVS model development (Zinoviev, 2005).

In another work, an executable framework based on UML Statecharts is developed by Mooney and Sarjoughian (2009). This work shows a subset of UML Statecharts models that conforms to certain properties, making them executable. In recent work, a Statecharts metamodel specialized for DEVS is proposed (Fard and Sarjoughian, 2015). Other works focus on defining UML state machines for behavioral definition along with use case, sequence, and timing diagrams (Risco-Martín *et al.*, 2009).

There is a significant distinction between viewing the behavior of atomic models in activities as opposed to Statecharts. We also think that activities as a language for specifying system behavior have not received sufficient attention, especially after the release of UML 2.5 (OMG, 2012). There have been few works that consider activities as a way to approach the simulation for models specified at higher levels of abstraction. They focus on transforming the higher-level models into an executable form. In (Foures *et al.*, 2012), the goal is to provide a simulation for activity diagram conforming to OMG SysML specifications. The solution was not developed for DEVS models (Pasqua *et al.*, 2012). Instead, it proposes transforming sequence diagrams to FD-DEVS models. It also generates Java code for atomic models that have simple behaviors. In the SysML profile for classic DEVS (Nikolaidou *et al.*, 2008), the activity diagram is used to facilitate the definition of the external transition function. There are also other three sub-diagrams for describing the behavioral specification of the atomic model. Those diagrams are SysML constraint diagram for defining states, parametric diagram for establishing state association, and state machine diagram for describing the internal transition, output, and time advance functions. In recent work, UML activity modeling is used to define the external, internal, output, and time advance functions for atomic models of a health care system (Ozmen and Nutaro, 2015). These models, as compared with mathematical specification or code, are more attractive to domain experts. However, the use of activities in this work

incorporates with the simulation routine resulting in agent-based models. Behavior modeling is achieved by representing the functions of the atomic and coupled DEVS model component of a system as UML activities. Defining their relationships takes place in terms of the DEVS simulator protocol.

Ptera (Feng *et al.*, 2010) is also another way of specifying an event-oriented model. It contains actions, final, and initial parameters that can attribute to an event that represents a vertex in the model. The activity has different notations for each one of those attributes. It lacks the rigorous definition of time as opposed to the Ptera model of computation. In contrast to the existing works, our focus is to employ activities and the activity metamodel itself and how to enrich it to arrive at activity-based DEVS model specifications. We will view the activity as a major diagram in the object-oriented paradigm to support the specification of an atomic model. However, rather than using UML activity modeling as-is, we are interested in specializing it to capture the atomic DEVS model syntax and conform to the DEVS execution semantics. The specialization is necessary since the UML activity syntax and semantics is aimed at satisfying a wide range of needs. Thus it lacks the essential constructs to specify atomic models that can conform to the DEVS formalism.

4.2 Approach

We begin by considering DEVS metamodels (i.e., DEVS to SMP2 (Yonglin *et al.*, 2009), MDD4MS (Cetinkaya *et al.*, 2011), and EMF-DEVS (Sarjoughian and Markid, 2012)) that have been proposed based on MDA. Based on such metamodels, we focus on the behavioral specifications using a model-driven approach to be consistent with the other existing approaches. UML State Machines (Nikolaidou *et al.*, 2008) provides a specification based on an object-oriented variant of the original Statecharts formalism (Harel and Politi, 1998). However, the metamodel of the UML State

Machines associates the *Behavior* element with the *State* and *Transition* elements. The behavior defined as an effect of *Transition* may also have actions assigned to it. Likewise, *Behavior* can be defined for *State* as an *entry*, *exit*, and *doActivity*. The action takes place in an activity as an executable node. A subset of UML 2.5 metamodel is shown in Figure 4.1.

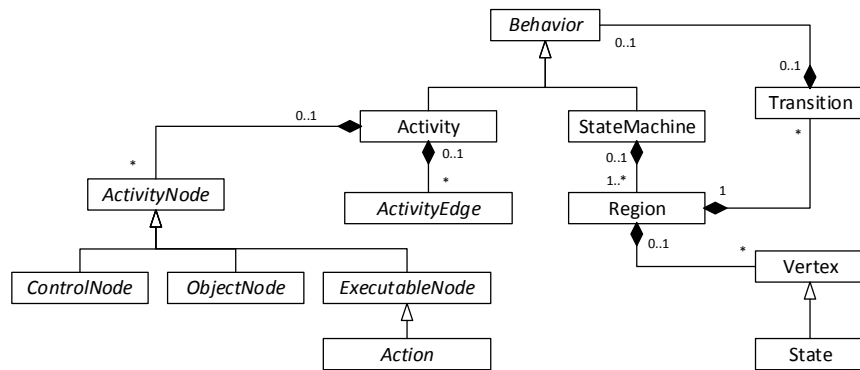
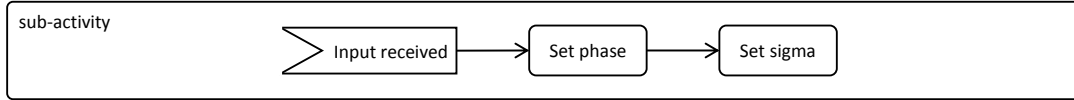
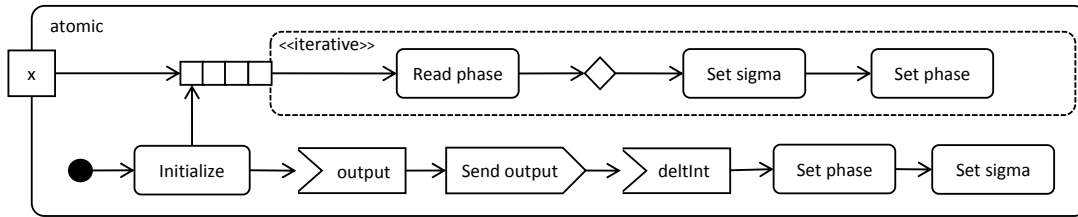


Figure 4.1: A Subset of Behavior Elements and their Relationships In UML 2.5 Metamodel.

Here, we examine how activities can be employed in the context of atomic DEVS modeling. Activities may be used from a different point of view in modeling behavior of an atomic model. A view may also depend on using other kinds of models. We consider three views for specifying the behavior using activities. The first view is to create a separate activity diagram for each routine belonging to a function. For instance, the activity captures the behavior, including actions and their order, for using an input to set the state of an atomic model. For each control state, there is a set of activities defined to handle one or more inputs. The second view is to create an activity for each function. For an atomic model, five activity diagrams are corresponding to each of the external transition, internal transition, confluent, output, and time advance functions. The third view defines one activity diagram that corresponds to the behavior of an atomic model as specified by all of its functions.



(a) A Sub-Activity to Describe a Simple Routine.



(b) One Activity for the Atomic Model.

Figure 4.2: Different Views of Activities DEVS Modeling.

4.2.1 Three Views for Specifying Atomic Model Behavior

In the first view, an activity is defined to specify the behavior for each event-routine as a subordinate unit of the corresponding DEVS atomic model component. We refer to them as sub-activities. It takes advantage of the existing definitions provided in the activity diagram to complement Statecharts by ordering actions. It does not involve complex behavior handling concerning other DEVS components nor the simulator. However, it can represent some basic patterns intrinsic to modeling, for example, the external transition function, as shown in Figure 4.2a. Other procedures are assumed to be handled externally in a separate activity or even in a different model that can communicate with the activity model. An example of this behavior would be managing decision points based on the control state for the external transition function. The selected activity is executed directly once being called. Thus, it is a suitable when those activities become sub-models due to their simplicity. The option removes most of the encountered complexity when handling the total state of an atomic model in state transitions.

The notion of activities provide some means for handling more complex scenarios among atomic model parts in addition to the capability of handling flows. We devised the second view to capture the concept of control in DEVS functions. An activity is created for the external and internal transition, confluent, output, and time advance functions. In this view, we use additional constructs for representing behavioral patterns that can be captured in the DEVS functions. For example, a decision node can be used to check for the current phase. Decision nodes can also be used to monitor ports and input values. Arbitrariness is presumed for any order that is not explicitly defined in the model; this conforms to the parallel DEVS formalism. For instance, the expansion region construct can be used to iterate a collection of received inputs in an arbitrary order. The execution of such a scenario is given by the simulator protocol and its implementation in some target simulator. These activity models, as compared with the first view, have more artifacts for handling more complex patterns. However, the modeler has more capabilities at hand in modeling and constraining the behaviors of atomic DEVS models.

Considering the functions of an atomic model to be viewed in one activity is also possible; this is the third view. However, this holistic view necessitates having definitions and artifacts involving relationships between the atomic model functions. Unlike the second view, implicit relationships, for example, between internal and external transition functions may be modeled. In this view, we may also take into account the simulation protocol. For example, in the DEVS-Suite simulator, the atomic models components are specified and also simulated independently of each other. In this view, the behavior of the model is defined explicitly in some structure that accounts for the simulator. Such a structure must conform to the simulator interfaces using some activity artifacts such as accept event actions. A major drawback for this view is that model specification is tightly dependent on the abstract simulator. The be-

havior of a simple processor model (Zeigler *et al.*, 2000) (it processes inputs it receives and sends out processed inputs) is shown in the first view Figure 4.2a and the third view (Figure 4.2b). Next, the second view is detailed along with the processor model shown in Figures 4.3 and 4.4.

4.2.2 Activity Specifications for Atomic DEVS Model

Considering all the UML activity constructs (OMG, 2012), we can model even more complex behavior by using a fork, join, decision, merge nodes, and expansion regions. We also use call behavior actions. The nodes are to specify the behavior of the atomic model components where the call behavior actions are to depict the communication points with other models. Each node is used to represent some specific concepts in the atomic models. We describe these artifacts and their corresponding concepts in the atomic DEVS model. Then, we present how these artifacts (see Table 1) can be used to define the behavior of the exemplary processor model with multiple inputs.

The fork node splits the flow into concurrent flows, each having multiple input events. The expansion region can also be used for the same purpose. Unlike the fork node, the expansion region is a specialized action which is a structured activity node. It defines concurrent flows of its included elements for the number of input events in the collection of received input events. However, the execution might be performed sequentially based on the simulation engine design and its implementation. The iterative expansion region processes the collection of received inputs sequentially. The order of processing these inputs is arbitrarily determined. In a parallel atomic DEVS model, multiple inputs can be received simultaneously. Therefore, the activity has a collection of inputs processed in an iterative expansion region. Further, these inputs can be examined inside the region via some activity nodes nested in the same

region. The activity can be terminated inside the region.

The decision node provides a means for controlling the flow by having multiple outgoing edges with guards assigned to all edges. For example, the decision node can be used to control the flow based on the phase. The phase is read in a preceding action and then evaluated by some guard conditions associated with some outgoing edges from the decision node. An *else* can also be defined as a guard for one of the outgoing edges.

We can also use a special kind of action called *CallBehaviorAction*. It allows the invocation of some behavior in a different model. We can use this action to call the behavior specified in the sub-activities discussed in the first view. For each input arriving in some control state, the sub-activity is called synchronously meaning that the main activity will not proceed until the called behavior completes.

We now create multiple activities for atomic model behavior, as shown in Table 1. These activities are created for modeling the behavior of an atomic DEVS model with multiple inputs. We model the external, internal transition, and output functions, each in a separate activity. Activity models for the time advance and confluent transition functions can be specified similarly. We use the artifacts as described in Table 1 to provide the required modeling capabilities for capturing the behaviors of atomic DEVS models. The semantics of these artifacts as defined in UML specification aligning with the concepts described for DEVS.

4.2.3 Action Specifications for Atomic DEVS Model

Actions are used as executable nodes in the activity diagram. They are the fundamental units of behavior specification in UML. Some actions change the state of the system. This kind of action in our approach satisfies how states can change in the DEVS atomic internal and external transition functions. For example, a phase

change is modeled using add structural feature value action named Set phase. The value of the target phase is modeled using value specification action with the value of the target state. The value, to be assigned, transmits through the output pin using an object flow connected to the value input pin of Set phase action. The actions can take place in the activities corresponding to the external and internal transition functions. A similar procedure follows for setting sigma or other state variables.

Other kinds of action support handling objects. *Read self action* is used to obtain the current object context and place it in its output pin. Value specification action is also used to provide specific value and put it in its output pin. The structural feature actions are used for either assigning or retrieving some features of an object. Both are used for the phase, although they can be used for other features or more complex objects.

Event actions can also be used. An accept event action can be used to model the waiting for an event to occur in some other entity to proceed in the activity flow. In the context of UML actions, this event can be caused by the simulation protocol to trigger some other parts of an atomic model. It may also trigger other models that decompose the behavior into multiple models. There can also be an *accept time event* action which is used to model wait time. We also use a send signal action which can be used to model invocation for some other components. It is also used to enforce some order when used with accept event explicitly. For example, to enforce the order of executing the output and internal transition function, a sending event signal must complete enabling the accept event action. However, this sort of scenario could be viewed as part of the simulation protocol.

We also use invocation actions. An example of these actions is *send signal action*. A *call behavior action* is used to call either an operation or an activity. The action can be synchronous if it has to wait for the called behavior to complete. Otherwise, it can

be asynchronous and then immediately proceed after calling the associated behavior or operation. We use synchronous call behavior action in the external transition activity to enforce the activity to wait for the completion of sub-activities before proceeding to other activities. Suspending is necessary since the behavior of the atomic model executes sequentially (i.e., the behavior in any of the functions of an atomic model is sequential).

Using the described activity diagram artifacts, we can create activities for the output and internal transition function for the processor model shown in Figure 4.3. The activity for the external transition function shows in the following section.

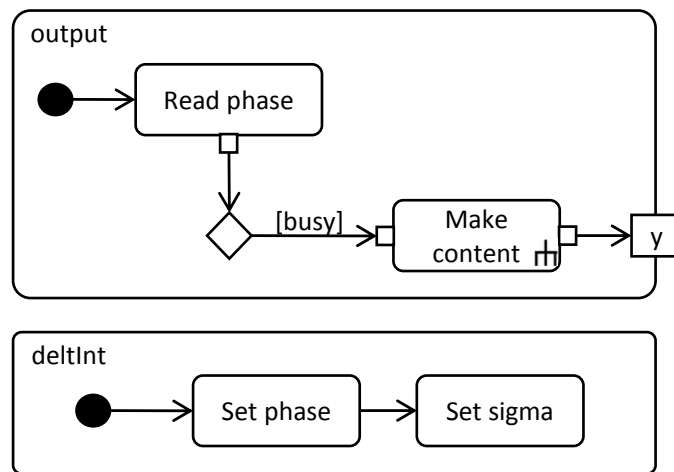


Figure 4.3: Activity Models for the Processor (Simplified).

4.3 Statecharts and Activities

The actions belonging to activities can be used with states and transitions in Statecharts. These actions can be viewed as the sub-activities in activity models detailed in the previous section. This approach follows the syntax of UML 2.5 where the effect of the transition is described using the Behavior element which is a generalization for the Activity as well as the State Machine (see Figure 4.1). We consider the DEVS

metamodel (Sarjoughian and Markid, 2012) as a higher-level abstraction to couple behaviors of atomic metamodel (Sarjoughian *et al.*, 2015). The proposed abstraction offers to couple their input and output ports as formalized in the parallel-coupled DEVS model.

Together Statecharts and Activity models support richer behavioral modeling for atomic DEVS models. Figure 4.4 illustrates this approach and highlights the basic relationships between Activity and Statecharts models. The DEVS structural meta-model for the atomic and coupled DEVS models is not the focus except the input and output ports for atomic models. The top left-hand side shows the coupled model GP (GeneratorProcessor) composed of a generator (Generator) and processor (Processor) models. The top right-hand side shows the external transition function of the exemplary processor in the activity model. The bottom left-hand side shows the external function of the same exemplary processor as a Statecharts. Finally, the bottom right-hand side shows one of the sub-activities. The sub-activity can decompose, such that, it includes other activities to develop simpler behavior models. That is, we avoid developing unnecessarily complex behavioral model specifications. The decomposition process must conform to the relationships with the Behavior element as defined in the UML 2.5 metamodel.

The Activity and Statecharts models show in Figure 4.4 to provide complementary behavioral specifications for the processor model. Both can be used to describe DEVS model specifications; none of these alone is known to be sufficient to have a complete specification of an arbitrary atomic model. Given the modularity in DEVS (models can only communicate with one another through couplings), the Activity and Statecharts models can only represent encapsulated behaviors of atomic models. These behavioral models capture different aspects of atomic models using distinct modeling syntaxes and semantics. We can use these behavioral models to concretely

represent the abstract mathematical specifications of the atomic model functions. In the processor Activity model, the inputs for the atomic model processor is defined as an input parameter for the activity. The input, however, is defined as an event for its corresponding transition in the Statecharts model. Both models define phase state transition from passive to active. In the activity model, we use multiple actions and control nodes to describe the implementation specified in the state transition. We use the read phase action, decision nodes, and set phase action. This sequence of nodes in the activity model is equivalent to state transition and actions defined in the Statecharts model.

Using Activity modeling overcomes some limitations in the other Object-Oriented modeling methods given its unique capabilities such as sequencing, control, and data flow. The use of DEVS Activity modeling is promising for generating models that can be executed in simulators. Such models are especially useful when considering the ongoing efforts to have tool support for defining model specifications and automated code generation. Overall, as Figure 4.4 shows, Statecharts and Activity models complement each other and support creating a richer specification for atomic models.

The activity diagrams in Figure 4.4 may be developed in tools such as Papyrus (Eclipse Foundation, 2016b). A modeling engine specialized for developing activity models for atomic DEVS models can also be developed. The coupled GP model and the external transition function belonging to the Statecharts of the Processor model are developed in CoSMoSim ACIMS (2017a).

4.4 Conclusion and Future Work

Few approaches are proposed to utilize the potential benefits of employing MDA concepts for DEVS. These approaches follow guidelines to enrich the development of simulation models. Including behavioral specifications in a model, the development

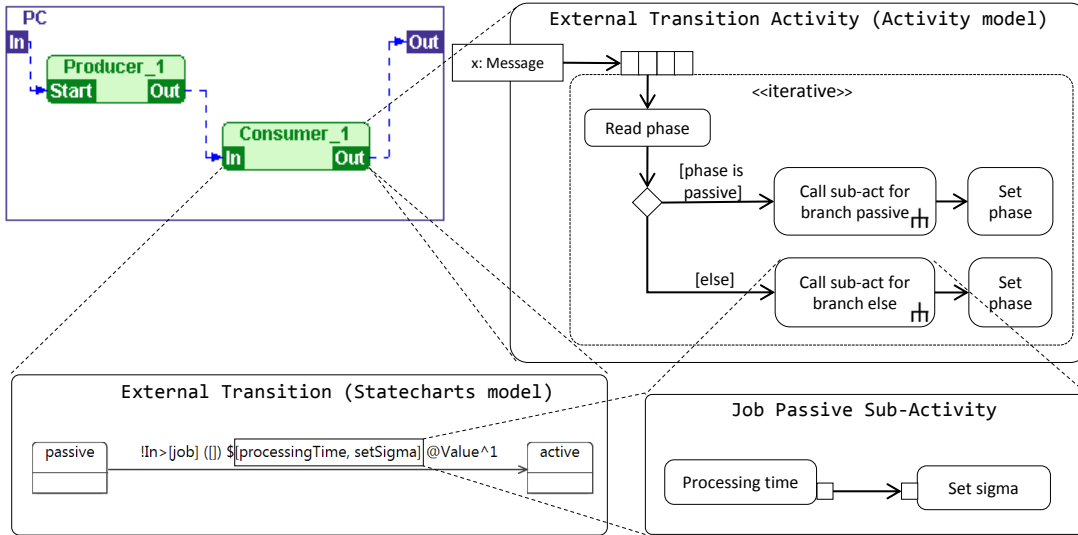


Figure 4.4: The Overall View of the Approach and Relationships between Different Models of Consumer Behavior.

lifecycle demands careful usage and guided by the MDA framework. Building models in stages can help modelers move across model abstractions. We can start developing Activity models and supplement them with Statecharts models, for example.

In this work, we described using modeling artifacts such as activity nodes and activity edges to specify behavior patterns such as sequencing and synchronization defined in Activities Behavior metamodel. We proposed customizing activity modeling to specify behaviors for atomic models. The activity metamodel can be used in various ways. We discussed creating activity models considering different views and exemplified behavioral activity modeling for an atomic processor model by a set of DEVS-based activity constructs. For future work, we plan to extend this work to support domain-specific activity-based behavioral DEVS modeling. We also plan to extend the CoSMoS to support Activity modeling for DEVS atomic models. The extension requires Statecharts and Activity models not only conform to the DEVS formalism but also consistent with one another. Achieving this kind of capability

may lead to improved model verification and validation.

DEVS SPECIFICATION FOR MODELING AND SIMULATION OF THE UML ACTIVITIES

The path toward utilizing the capabilities available in discrete event system modeling is essential. We are currently still far away from fully enhancing the use and therefore preventing failures that may arise in such systems. The concern of advancing and investigating discrete event systems has been continuously growing to attain an appreciation of the potential solutions and consequently taking advantage of them. UML has been dominant in the world of software modeling. Some aspects of it bear resemblance with discrete event systems which have been subject to research in the last decade from two points of view. The first one is to use UML as a language for system specifications while the other one is to supply UML with concepts to overcome its weakness in terms of formal grounding. The potential value that can be added to UML by providing a rigorous mathematical specification is priceless. It brings the value of formal specification into a widely used language by modelers.

However, there is a substantial complexity that can arise while trying to bring the formal specification to such a language. First, we are dealing with a high level of ambiguity that is necessary to maintain some level of generality. The language has standardized around the concept of allowing end-users to comprehend its models in a human-understandable manner. Adding formal specification may potentially result in relatively sacrificing some general ideas and definitions. Such exclusion certainly makes the problem challenging in addition to the inherent complexity and ambiguity that might arise in the language itself.

The problem has been widely discussed in research from both theoretical and

practical aspects. The solutions also widely vary. Researchers have been continuously providing extensions, frameworks, and tools. Some solutions rely on further knowledge about the intended domain while some others attempt to remain more neutral and domain-independent. The same perspective has also applied regarding platforms and applications. The notion of behavioral DEVS metamodeling is introduced in a previous work (Sarjoughian *et al.*, 2015) to provide some support for behavioral specification at the meta-levels. SysML is one of the most common UML profiles that has been devised based on similar motivation for system engineering, and yet it is challenging to simulate (Nikolaidou *et al.*, 2016).

Some current solutions do not incur the current lack of formality. Others require a substantial amount of additions and extensions to resolve some ambiguities. On the other hand, many approaches take a different direction and address the problem from an implementation standpoint, for example, by operational semantics for well-formed models. The Model-Driven Architecture (MDA) as well has been extensively used to address the problem from an architectural point of view.

A suitable candidate to be used for activities is Parallel DEVS (Chow, 1996). In so doing, the fundamental activity behavioral elements of UML (OMG, 2012), which are mainly action nodes in activities, can map to Parallel DEVS based on their resemblance with atomic models. Elements of this subset may have inputs and outputs. The transition function can be then devised for each element to implement the behavioral specification according to their semantics. For example, the semantics of the fork node can be specified. This realization can result in a conveniently visual representation and simulation of surface UML models that have a collective behavior in terms of its foundational elements such as in fUML (OMG, 2018). This is achieved by utilizing a Parallel DEVS simulator such as DEVS-Suite (ACIMS, 2017b).

In this work, we begin by giving some background about related matters of UML

activities and the foundational UML subset (fUML). We then discuss the related works. Next, we describe the basis for our rationale about the concept of activity modeling and simulation and what does it mean in terms of DEVS modeling. Then, we establish a fundamental ground for mapping between activities and DEVS concepts. We demonstrate the approach with a simple illustrative example. Finally, we discuss some findings and steps toward continuing the work in both the near and long term future.

5.1 Related Work

A significant effort has been continuing to enhance the process of model-driven practices for modeling in general and simulation modeling as well. The notion of allowing models to be executable has existed for a while. In (Harel and Gery, 1996), an integrated set of languages are developed for object modeling around statecharts. The goal is to produce an executable model which cannot be achieved without defining a precise semantics. That follows by an attempt to define a formal operational semantics for UML statecharts (Latella *et al.*, 1999). Another executable UML has been introduced by (Mellor *et al.*, 2002) to complement UML with the code to make it executable using model compilers. The notion of the proposed executable model is inspired by approaches such as (Stahl *et al.*, 2006). In (Kirshin *et al.*, 2006), a UML simulator is defined based on a generic model execution engine. The simulator relies on the available knowledge in the model upon the start of the simulation. It suspends when there is missing information via user/tool interaction. More recently, the fUML (OMG, 2013) is proposed to provide the semantics necessary for executing a subset of UML. Mayerhofer (2012) used fUML to enable model testing and debugging. These capabilities become accessible in a model execution platform called MOKA within the Papyrus Eclipse (Eclipse Foundation, 2016b).

There has also been an effort to employ model-based and model-driven methodologies for the system-theoretic specification (Risco-Martín *et al.*, 2009; Mittal and Martín, 2013a). Mooney and Sarjoughian (2009) utilized DEVS for the creation of executable UML models based on statecharts. This approach, unlike other DEVS-based approaches, is grounded by providing both specification syntax and execution semantics with well-defined timing. Such grounding is necessary for concurrent handling of events when developing composite executable UML models.

Activities have been a major modeling approach to resolving some limitations in the current modeling practices. We characterize these efforts based on their purpose. Some efforts aim toward automating and producing models that are suitable for production, where the others are built based on the theory of modeling and simulation. The model is a foundational element from both perspectives. The latter can also be considered as a theoretical basis for general system design instead of just being tied to simulation purposes. Thus, on one end, this is an attempt to devise a methodology based on the DEVS formalism for activity modeling. On the other end, it provides a profound means for specifying their precise semantics.

Our methodology relies on a different perspective in approaching model execution. We consider that the creation of the activity model is conducted for simulation purposes even though it is intended and often used for actual software product development. We want to ensure that the models are established based on rich system-theoretic specifications. At the same time, we also maintain the same capabilities by debugging techniques such as visualizing, controlling, and tracing the execution yet in a disciplined manner. We take a similar position with Risco-Martín *et al.* (2009) and Mooney and Sarjoughian (2009). However, as in our previous work (Alshareef *et al.*, 2016), we keep our focus on the activity modeling and try to achieve our new target goal exploiting the DEVS formalism.

5.2 Activities Simulation Through DEVS: Finding Rigor

There is some level of difficulty when it comes to modeling for simulation. A profound simulation for activities accounts for further knowledge and details beyond basic debugging capabilities provided in some approaches and tools. Although there are some temporal aspects in the process of model debugging (Mayerhofer, 2012), the notion of time is not explicit in the debugging modes. The step is intrinsic to the simulation modeling of dynamical systems based on a more expressive time notion. In DEVS, the time period assigned to any state change due to external and internal transition functions has arbitrary accuracy. Furthermore, the construct called elapsed time allows inputs to be handled by the external transition function at any future arbitrary time instance. These definitions can be effectively utilized to provide a stronger foundation for the simulation of activity modeling. Instead of using a step-wise or breakpoints as mechanisms to handle the execution of the activity model, the time advance function and the notion of sigma are used. The step can then take place during runtime based on these definitions. Figure 5.1 shows an overall view of how the concepts of modeling and simulation employ in current practices. We will make use of these entities as defined by Zeigler *et al.* (2000) to better perform the task.

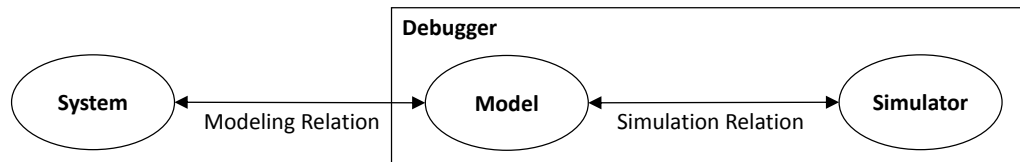


Figure 5.1: A Simplified View of Employing Concepts in M&S for Activities Modeling.

We construct the activity simulation based on the hierarchical and modular simulation framework for the DEVS simulator. A set of atomic models generally specifies the basic activity constructs. Consequently, each construct can be then simulated.

The coupling takes place between atomic models. The activity model is collectively defined via coupled models. The communication between elements is handled through messaging to represent the control as well as the object flow. The locus is transmitted to other components according to the semantics of the activity.

5.2.1 *A DEVS Grounding for UML Activities*

This subsection presents how UML activities are treated from a DEVS standpoint, including their structural and behavioral properties. A mapping is proposed to facilitate the process of understanding the bridging points between activities and the DEVS formalism. The mapping includes the general constructs and more concrete constructs thereof. We also discuss the modularity and the generality of the mapping in subsequent sections. We shall begin with the activity nodes and then consider the edges.

The activity node, which is the most abstract node element in activities, is generally specified by an atomic model. Most of the specialized activity nodes bear resemblance in terms of their structural properties as opposed to their behavior. Therefore, the specification of the atomic model needs to be specialized further for the atomic model behavior to define the semantics of the corresponding activity node. The action, for instance, is treated as an atomic model with some input and output ports (see Figure 5.2). The behavior of the atomic model is defined by the (external, internal, and confluent) transition, output, and time advance functions. It is defined as the most concrete element in the current activity hierarchy. The activity node specializes as control, object, and executable nodes, among others. These elements also specialize further. For example, the decision node is a subtype of the control node. Its semantics are defined in the behavior of the atomic model that corresponds to the decision node. Although its structural properties are set at a higher level,

since all elements share the same structural characteristics. The expansion region is a particular case of the activity node where the coupled model is used for it. The reason is quite straightforward since the expansion region may contain multiple nodes and edges, which in turn map to their corresponding atomic models. The elements of any activity are unique and may also have unique relationships to other elements in the activity. This abstraction exactly mirrors that of flat DEVS coupled model specification.

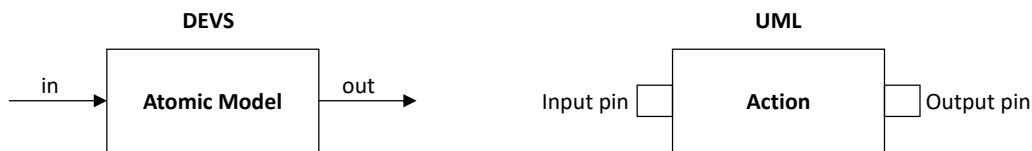


Figure 5.2: The Action, Which Is a Special Type of Activity Node, Is Treated as an Atomic Model with Some Input and Output Ports.

The internal coupling is used to specify the activity edge, which is a supertype for the control as well as the object (data) flows. The input and output ports make modeling more accessible and therefore used to specify the input and output pins alongside with ports to handle the flow. The activity node has at least one input port, which is used to enable it. Note that, Parallel DEVS is selected since it allows for receiving input (event) values simultaneously via multiple input ports. The nodes vary in the number of their input ports based on their concrete types and incoming edges. The multiplicity is, in a sense, similar to the activity nodes in having input or value pins, or not having any. The output ports are treated similar to inputs except they account for the outgoing edges and output pins. We note that, at a minimum, a non-trivial atomic model must have at least either external or internal transition state transition, two outputs, and two state variables. One variable represents the assignment of time duration for operations. The other one represents at least two

values for the model to be in (Wymore, 1993).

Thus, the previous components can serve collectively to form an activity model. The model is viewed as a coupled model from a DEVS perspective. Edges establish the connection between different activity constructs. In the DEVS formalism, the specification of the external input and output interfaces, components, and the coupling relation are included to serve as a means for establishing models from yet other DEVS models (Zeigler *et al.*, 2000). The external input coupling specifies the connection from the input parameter in the activity and considers it as an external input. It conforms to having two distinct components as required by the Parallel DEVS coupling legitimacy property. The coupling connects to some component ports. The external output port is also used in the same manner but for the activity output. The internal couplings are used as discussed in the previous paragraph to specify edges. Table 5.1 shows a subset of the mapping, although additional elements might be needed to put the activity in a purely modular object-oriented modeling context. For example, internal couplings can take place for communication between objects. Some object node can request this communication in the activity model, and then outputs are sent out to other actions accordingly. Other couplings are used for controlling the model concerning the activity semantics. In the UML activity, it is not necessary to require two components to communicate via ports strictly. That is, a component can invoke operations of some other component instead of using signals (messages).

5.2.2 *The Semantics of Activities*

The basis of any simulation environment for activities has to account for execution semantics. The objective is to define semantics that is specific to DEVS modeling and yet general in the context of activity modeling. The activity initializes by either an initial node or some external influence. For example, the activity can reside in

Table 5.1: A Subset of the Mapping for Activity Elements

| Activity | DEVS |
|---------------------|--------------------------|
| Activity | Coupled model |
| Activity node | Atomic model |
| Expansion region | Coupled model |
| Input and value pin | Input port |
| Output pin | Output port |
| Activity edge | Internal coupling |
| Activity parameter | External input coupling |
| Activity parameter | External output coupling |

the context of some other model such as a class. Any valid flow can represent an execution of a particular activity performed by a course of action. The control nodes manage that flow; however, they do not impose changes on the associated objects. Each control node, as well as to object and action nodes, has its semantics. Our objective is to define their semantics formally in a set of DEVS models. For instance, the semantics of a decision node can be defined in the behavior of its corresponding atomic model. Upon input arrival, the phase is changed to "*executing*" to denote the existence of an active node in this particular execution path. Then, in the next time step, the condition associated with this specific decision node is evaluated.

The result of the evaluation will determine via which port the output sent out. If the assessment of the guard condition is true for more that one case, the output will be sent out via one of the output ports arbitrarily. It is, however, the modeler's responsibility to account for such a scenario if a mutual exclusion is required, for instance. Finally, the phase is set back to passive by the internal transition function.

The formal specification of the atomic model that corresponds to that is defined in Parallel DEVS as

$$\begin{aligned}
DEVS_{processing_time} &= (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta), \text{ where} \\
IPorts &= \{“in”, “in1”\}, \text{ where } X_p = V \text{ (an arbitrary set);} \\
X_M &= \{(p, v) | p \in IPorts, v \in X_p\} \text{ is the set of input ports and values;} \\
S &= phase \times \sigma \times condition \times store, \text{ where} \\
phase &= \{“passive”, “executing”\}, \sigma = \mathbb{R}_{0, \infty}^+, \\
condition &= \{true, false\}, store \in X_M; \\
OPorts &= \{“out”, “out1”\}, \text{ where } Y_p = V \text{ (an arbitrary set);} \\
Y_M &= \{(p, v) | p \in OPorts, v \in Y_p\} \text{ is the set of output ports and values;} \\
\delta_{int}(phase, \sigma, condition, store) &= (“passive”, \infty, condition, x) \text{ where } x \in X_M; \\
\delta_{ext}((phase, \sigma, condition, store), e, X_M) &= \\
&((“executing”, processingTime, !condition, (p_1, v_1), \dots, (p_n, v_n)) \\
&\text{ if } p_i \in \{in, in1\}, i \in \{0, \dots, n\}; \\
\delta_{con}(s, ta(s), x) &= \delta_{ext}(\delta_{int}(s), 0, x); \\
\lambda(“executing”, \sigma, condition, store) &= (out, store.v) \text{ if } condition = true \text{ and } store.p = in, \\
&= (out1, store.v) \text{ if } condition = true \text{ and } store.p = in1, \\
&= (out, store.v) \text{ if } condition = false \text{ and } store.p = in1, \\
&= (out1, store.v) \text{ if } condition = false \text{ and } store.p = in, \text{ where } (p, v) \in \\
X_M; \\
ta(phase, \sigma) &= \sigma.
\end{aligned}$$

The processing time is defined as an abstraction to represent the step-wise execution of the activity model. The other control nodes are defined similarly to the decision node while distinguishing between their unique structural and behavioral properties. The fork and join nodes are for branching in and out the flow with a

synchronizing capability. That is, the join node waits for all incoming flow loci to transition to an executing state and the fork sends out loci via all its outgoing flows. The initial node does not have an incoming flow, and therefore, no input ports shall manifest for this purpose. Similarly, the final node does not have an outgoing flow. It should be noted that any output from an atomic or coupled model automatically duplicates per number of couplings that it has to output to. The duplication of outputs takes place according to the external input and output couplings as well as internal couplings.

5.3 Network Switch: an Example

Grounding activity models into DEVS has been accomplished at the meta-layers. Thus, the process of creating concrete models becomes easier. We choose the network switch for several reasons. It can be specified as a coupled model. It illustrates the handling of inputs in Parallel DEVS. It also exemplifies inherently different behaviors since it contains a switch as well as a processor. The model is described in (Zeigler *et al.*, 2000) as shown in Figure 5.3. The internal transition function applies before the external one in the case of having both the switch and one processor imminent. The switch decides to send out the job via one of its output ports based on its polarity. It does either one of the following scenarios. The first scenario, incoming input from the first input port gets directed to the first output port. Also, inputs incoming from the second input port gets directed to the second output port. The second one, if it is on the other polarity setting, it reverses the first scenario. The nature of the switch in its general form resembles the semantics defined for the decision node of activity. From another perspective, the decision node can be considered as a higher level of abstraction of the switch. Therefore, the activity model in Figure 5.4 can be seen as a higher-level abstraction of the coupled model.

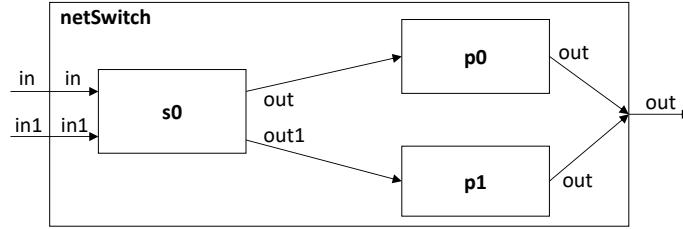


Figure 5.3: The Network Switch Parallel DEVS Coupled Model.

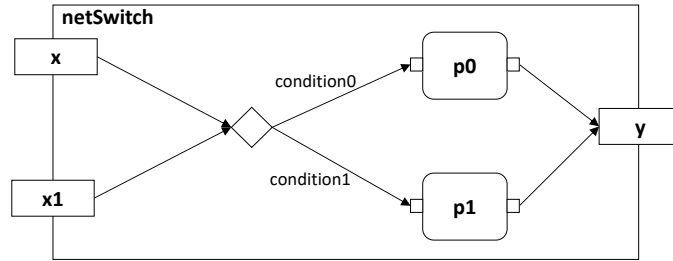


Figure 5.4: An Activity for the Network Switch Coupled Model.

In addition to the abstraction, the semantics of activities are incorporated as well. The behavior of the switch accounts for the semantics of the decision node. In other words, the polarity and inputs are checked as conditions. Once evaluated, the node will send out the job to a corresponding processor. Actions are treated as atomic models in general. However, they are treated as a processor in this example. This treatment is due to the processor behavior, which accounts for the semantics of the action. In other words, the semantics of activities are specified in the set of atomic models that are then underpinned by the semantics of the simulation protocol for the DEVS formalism.

5.3.1 Modularity

Thanks to the closure under coupling property, we can ensure the feasibility of constructing a hierarchical model based on the elemental constructs. The mapping is established based on the most abstract activity constructs, and then the behavior is

specialized accordingly. The most concrete elements are mapped into atomic models and used in an activity or expansion region via coupled system specifications. The coupled DEVS specification for the network switch activity is

$$A = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC),$$

where

$$InPorts = \{“in”, “in1”\},$$

$$where \quad X_p = V \text{ (an arbitrary set)}, \quad X_M = \{(v) | v \in V\};$$

$$OutPorts = \{“out”\}, \text{ where } X_{out} = V, \quad Y_M = \{(“out”, v) | v \in V\};$$

$$D = \{DecisionNode0, Action0, Action1\};$$

$$M_{DecisionNode0} = DecisionNode; M_{Action0} = M_{Action1} = Action;$$

$$EIC = \{((Activity0, “in”), (DecisionNode0, “in”)), ((Activity0, “in1”), (DecisionNode0, “in1”))\};$$

$$EOC = \{((Action0, “out”), (Activity0, “out”)), ((Action1, “out”), (Activity0, “out”))\};$$

$$IC = \{((DecisionNode0, “out”), (Action0, “in”)), ((DecisionNode0, “out1”), (Action1, “in”))\}.$$

Since this is also a system specification itself, it can be used further in a broader context within other system specifications. This also stands to provide additional benefits in the context of UML. However, more investigation on the mapping has to be carried out, given the current system specification that corresponds to only activities and their substances only. Various behavioral and structural subsets of the UML metamodel need to be investigated to determine how they can be treated in this broader context.

5.3.2 The Generality of the Models

Assuming the simulation protocol is domain-agnostic, we think the DEVS models that are built for the execution semantics of activities are also domain-agnostic. The behavioral specification is polymorphic in the sense that they capture the behavior that accounts for multiple types and values. The notion of specifying structure at the meta-levels is well established but not for the state transition behaviors (Sarjoughian *et al.*, 2015). We think of the created DEVS models along with their behavior to be situated at the M2 layer in the MDA. Their generic behavior can be used to simulate any specific instance. Despite that they are extensible, their current behavioral specification is sufficient to be applied to well-formed instances at the concrete level M1. A view of the simulation for some of the high-level activity constructs shows in Figure 5.5.

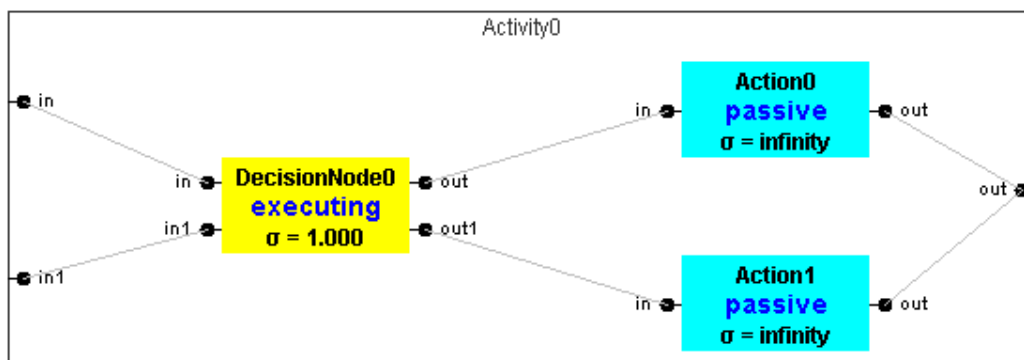


Figure 5.5: A Simulation View for the High-Level Activity Constructs Used to Model Network Switch (Implemented in DEVS-Suite).

Another important aspect of the models is regarding metamodeling; that is, the behavior of these models can be viewed as activity models. One activity corresponds to each function of the atomic model Alshareef *et al.* (2016). Such correspondence is due to the existence of the meta-layers and the conformance relationship between

them. The models that encompass the semantics of the activities can be viewed as activities. However, they are currently realized in Java code snippets that can take place in the DEVS-Suite simulator. The representation of these snippets in activity notation can be thought of based on the Annex A provided within fUML specification (OMG, 2013).

5.4 Future Work

We are currently working on the development of two packages for DEVS-Suite for full support for the simulation of any well-formed activity model. One package is to encompass the semantics of different control nodes as well as actions without losing generality. The second package is for interpreting activities as a previous step before simulating them.

The activity package contains the activity metamodel as discussed. It also contains generic code snippets for the semantics of each control node. The action is treated as a general construct. Additional research is yet needed to support further elaboration for specific types. The research includes specific concrete details of the action type as well as a mechanism to incorporate it with the current specification.

The interpreter is also added to make it easier to incorporate an activity model from the activity package standpoint. A certain checking has to take place to ensure the injected models are well-formed. We can benefit from that by eliminating as much of the code portions possible for target execution platforms. The transformation to an executable form is restrictive, particularly from the standpoint of automatically generating code for behavior specified in atomic models (Sarjoughian and Elamvazhuthi, 2009; Sarjoughian and Markid, 2012; Seo *et al.*, 2013; Mittal and Martín, 2013a). Thus, the potential of the interpreter in further automation of the process while having some control over the abstraction levels can be promising. We are working on this

issue, and our target objective is to ultimately provide these capabilities and make them accessible to modelers who use simulators such as DEVS-Suite. The direction of the transformation in a general sense remains open for future research.

5.5 Conclusion

In summary, we proposed a DEVS specification for UML activities in conjunction with the concept of executable modeling. The objective is to obtain a rigorous grounding for the modeling and simulation of activities based on system theory. We proposed mapping for activity elements into DEVS models. The mapping centers around the concept of activity nodes, including actions since they collectively serve as a basis for the activity and connecting them via edges. Also, we complemented that with the definition of their semantics. We demonstrated the approach by developing some examples of atomic and coupled models for a network switch according to the Parallel DEVS formalism. We also discussed some remarks regarding the modularity and expressibility of the models.

The research on behavioral modeling remains quite challenging. We employ a variety of concepts, frameworks, methodologies, and tools in a manner that is consistent with formal model specifications. The specification is important to be sufficiently powerful to account for non-trivial dynamical systems and their models. Approaching the creation of executable modeling from a modeling and simulation standpoint is useful. The value of enabling rich and mature concepts such as experimental frame (Zeigler *et al.*, 2000) can be achieved whenever possible by techniques that can help in the movement from different modeling layers to simulation and execution.

Introducing simulation, as opposed to debugging and testing, accompanied by its full power to the UML activity modeling, leads to benefits. Among these benefits is enabling underlying simulators to be employed for studying models during their

development life cycle. Such a goal is difficult to achieve without having precise model semantics. The DEVS formalism can be utilized to define these semantics. It is a suitable candidate to accomplish this objective. Moreover, it establishes the notion of time yet more rigorously which can be utilized further, for example, for cyber-physical systems and more broadly Internet-of-Things. The defined DEVS specification for the semantics can effectively serve as a basis for the concrete counterparts allowing a wide variety of component-based simulators.

ACTIVITY-BASED DEVS MODELING

Constructing simulation models, despite being quite costly and complex, remains indispensable and highly beneficial, especially for systems that do not lend themselves to analytical methods. System dynamics need to be determined in enough detail to sufficiently address its different aspects under study to attain the potential benefit ultimately. The models have to be then realized in certain computational and physical environments to enable the simulation and experimentation after that. As system complexity grows, so does the importance of behavioral modeling. There are existing concepts and techniques where the structure modeling can be handled systematically to account for further system complexity and growth. However, these techniques fall short concerning behavioral modeling. Increasingly, behavioral models are becoming large and therefore difficult to understand, formulate, and maintain using conceptual (informal) and mathematical modeling as well as their implementation in programming languages and evaluations.

The Discrete Event System Specification (DEVS) formalism (Zeigler *et al.*, 2000) can effectively serve as a basis for simulation-based design and formulation of modular, component-based system models. The parallel DEVS formalism, based on time, input, output, states, and state transition, is widely supported by simulators, DEVS-Suite (Kim *et al.*, 2009) for example, that have been implemented in different computing environments. Simulators need to serve different needs through advanced complementary capabilities, including action-based behavior specification. The input for simulators is mainly models, although the simulators significantly differ in the form by which the models have to be formulated, simulated, and evaluated. This

has led to the rise of utilizing the so-called Model-Driven Engineering in simulation, especially for the focus on creating platform-independent models. Models of this nature are less inclined to carry details specific to the execution environments and therefore can be used and maintained for a broader set of M&S platforms, a key benefit of Model-Driven Architecture (MDA) (Soley and the OMG Staff Strategy Group, 2000). The definitions in this approach have been proven to be consistent with the theory of modeling and simulation in multiple occasions (Risco-Martín *et al.*, 2009; Cetinkaya *et al.*, 2011; Sarjoughian and Markid, 2012; Mittal and Martín, 2013b; Sarjoughian *et al.*, 2015). In fact, it is desirable to have models of this nature. It allows modelers to focus on the problem and solution specifications in more neutral terms regarding specific details of simulation frameworks. Behavior specification is not as simple as structure specification, especially when system dynamics require understanding and formulation beyond conditional state changes and event handling. This fact is accounted for the recent approaches that adopt some of the other languages and formalisms (e.g., UML state machines) with capabilities that can afford to specify complex behaviors.

While some behavioral languages are adopted for the specification of DEVS atomic models, they significantly differ in their suitability, complexity, and provided capabilities. In this work, we attempt to dig deep into the activity modeling as a major modeling approach for the DEVS atomic model and by extension coupled model behavioral specifications. The approach is gaining more attention, given its promising prospects for enhancing system modeling in multiple domains. Activity metamodel has also undergone major advances in the recent decade especially the release of the UML 2.0 (OMG, 2005) and the foundational subset of UML (fUML) (OMG, 2013). The idea essentially is to adopt the UML activities for the behavioral specification of the DEVS atomic model according to the state of the art standards. Activities provide

some unique capabilities for other behavioral diagrams. We want to leverage them in a way that shortens the distance between the concrete models and their mathematical abstractions. The handling of actions in the activities gives a premise to overcome some of the behavioral modeling difficulties in general and the ones encountered in the other behavioral approaches (e.g., finite-state machines). A richer specification can be achieved when modelers consider a variety of behavioral specifications that better serve their needs.

We will discuss some necessary background on this subject. We will also compare and contrast our contribution with some of the existing approaches toward meeting the need for expressive behavioral modeling. We elaborate on the selected approach based on previous work (Alshareef *et al.*, 2016) in conjunction with further discussion and alignment with the DEVS formalism. A modeling engine is created as manifest to implement alongside the processor with the queue model as an exemplar.

6.1 Related Work

Researchers have employed the notion of behavioral modeling for different purposes. It has been used to bridge some distance between the mathematical specification of the system-theoretic model in the DEVS formalism and their counterpart incarnations in computational forms. We characterize these efforts in two major categories. The first one is a state-based approach, such as statecharts, while the second is a flow-based approach, such as activities. Both approaches are considered to be primary behavioral diagrams in UML 2.5, which we attempt to align with as much as possible without compromising rigor. The state-based approach has been employed by many (Schulz *et al.*, 2000; Zinoviev, 2005; Mooney and Sarjoughian, 2009; Risco-Martín *et al.*, 2009) to support defining behavior in the form of statecharts or state machines. These add new means to identify behavior for DEVS atomic models.

Furthermore, this approach provides graphical model development and mechanisms for validating models without execution, for example, using EMF (Sarjoughian and Markid, 2012; Fard and Sarjoughian, 2015). Another related work is Action-Level Real-Time (ALRT) DEVS modeling. In such work, the external and internal transition functions are formalized not only for the state but also actions. Actions have their timings and therefore executed using a real-time simulation protocol (Sarjoughian and Gholami, 2015). However, this variant of the parallel DEVS formalism does not account for the UML metamodel concept and the activity modeling.

Since some existing methods (statecharts and state machines) are aimed at certain aspects of behavioral modeling, the use of other methods becomes quite compelling. By doing so, it explains the tendency to use activity modeling as an approach to aid the development of DEVS models. The use of activities for the DEVS atomic functions $(\delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ complements other behavioral specifications with certain capabilities such as ordering and the containment of actions within a UML context. Also, there have been recent advances, especially for model execution using the foundational UML subset (OMG, 2013). Activity modeling is also used for SysML models (Nikolaidou *et al.*, 2008; Foures *et al.*, 2012) as well as developing model abstractions for application-specific domains (Ozmen and Nutaro, 2015). In the latter, activity modeling is purposed for simulation protocol, but not, for example, in terms of actions and controls within atomic models.

In this work, we consider activities as a complementary modeling approach for the visual behavioral specification of the parallel DEVS atomic model. We specialize in the activity metamodel to serve as a basis for action-level behavior modeling as well as proposing and developing a modeling engine for parallel DEVS atomic modeling (see Section 6.3). The activity models are graphically developed utilizing the capabilities provided with Eclipse Modeling Framework (EMF) and its Graphical Modeling

Framework (GMF). We create an Ecore metamodel to encompass the defined elements and constructs according to the activity modeling approach. Our objective is to account for as many details as possible of behavioral specifications relative to simulators. At the same time, we consider the principles defined in the Model-Driven Development and Model Driven Architecture. Such principles enable the approach to expand further to incur the expected complexity that may arise during the model development for behavior. The ability to come across some level of control over the different levels of abstractions in such a grounding can be highly beneficial.

6.2 Activity-based DEVS Modeling Approach

We thoroughly investigate the use of activities in DEVS modeling starting by bringing into a common perspective the DEVS formalism, behavioral specification, and UML behavioral diagrams. We show where activities can come into place within existing DEVS metamodels (i.e., DEVS to SMP2 (Yonglin *et al.*, 2009), MDD4MS (Cetinkaya *et al.*, 2011), and EMF-DEVS (Sarjoughian and Markid, 2012)). We also show where they can come into place by benefiting from a model-driven approach and considering the other behavioral diagrams, specifically state machines. Then, we discuss possible views to employ activities for DEVS modeling and then elaborate on these views to establish the selected approach.

The functions δ_{ext} , δ_{int} , δ_{con} , λ , and ta are defined in a DEVS metamodel (Sarjoughian and Markid, 2012). These functions capture both the abstract structure and behavior of the formal DEVS atomic model specification. Each function has one or more elements defining its structure. For example, the structure of the external transition function is defined strictly only to have input events and state. On the other hand, the structure of the output function is defined to have only output events and state. The behavior of these functions defines what in the model can be changed

(allowed behaviors). For example, in the external transition function, the state of the model can change, but in the output function, the state change is not allowed. Therefore, the behavioral specifications can be defined at the meta-layer to represent a system's behavior, for example, state machines. The activities can also be used to define the syntax for the functions mentioned above. In the UML state machines, a variant of the original statecharts formalism (Harel and Politi, 1998), the meta-model of the UML 2.5 associates the *behavior* element with the *state* and *transition* elements. *Behavior* defined as an effect of *transition* may also have actions assigned to it. Likewise, *behavior* can be defined for *state* as an *entry*, *exit*, and *doActivity*. The action takes place in an activity as an executable node, which is a subtype of the *ActivityNode*. According to the subset of UML 2.5 metamodel in Figure 4.1, the activity can be used to define behavior. This behavior, therefore, can either be super or be nested in some other behavioral model such as state machines. For example, the activity can be associated with the transition to define the effect of it as discussed in Section 2.3.3. However, in this work, we focus on the use of activities, although various views exhibit.

The *Activity* can be viewed to collectively represent the behavior of the atomic model or some of its constituents thereof. We ignore the holistic view of activities for the whole atomic model since that imposes some of the ordering relationships among the different functions which must be handled by the simulation protocol. Thus, we consider each function separately by devising an activity for each one and design activities for the subordinates thereof. The advantage of creating activities for the subordinates is to allow them to compose in various ways with potentially different kinds of models. For example, they can be contained within other activities as an action for external behavior or within state machines as an effect for a transition.

We now create multiple activities for the atomic model specified behavior by

each function. These activities are created for modeling the behavior of an atomic DEVS model with multiple inputs. We model the external, internal transition, and output functions, each in a separate activity. Activity models for the time advance and confluent transition functions can be specified similarly. We use the modeling elements as described in Table 6.1 to provide the required modeling capabilities for specifying the behaviors of atomic DEVS models. The semantics of these elements, as defined in the UML specification, have been aligned with the DEVS concepts.

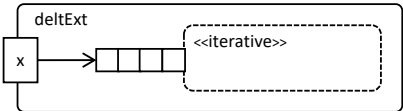
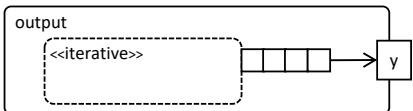

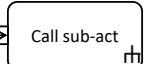
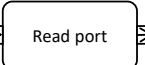
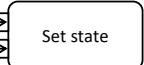
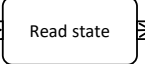
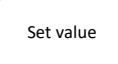
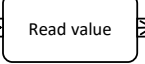
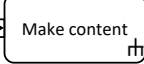
| | | | |
|---|---|--|---|
|  | <p>A template for an activity diagram for the external transition function receiving a collection of inputs and processing them in iterative expansion region</p> | | |
|  | <p>A template for an activity diagram for the output function with an output parameter node</p> | | |
|  | <p>A template for an activity diagram for the internal transition function</p> |  | <p>An action to call behavior specified in another activity</p> |
|  | <p>Read structural feature action to read the input and assign it to the output pin</p> |  | <p>Receive a value and assign it to the phase</p> |
|  | <p>Read structural feature action to read the phase and assign it to the output pin</p> |  | <p>Set value and assign it to the output pin</p> |
|  | <p>Read value of the received feature via the input pin</p> |  | <p>An action for making content to be used for preparing output</p> |

Table 6.1: Activity Specifications for Atomic DEVS Model

The activity diagrams in Figure 4.4 depicts some of the behavioral specifications of the processor model (a richer formulation to be demonstrated in Section 6.4) as well as a holistic view of the devised approach. The activity in the upper right corner depicts the behavior of the external transition function, while the activity in

the lower right corner represents a subordinate unit for a state transition. Therefore we call them sub-activities. A modeling engine specialized for developing these activity models is developed and will be discussed in the following section. A coupled Producer-Consumer (PC) model (upper left corner) and the external transition function belonging to the statecharts of the processor model (lower left corner) are developed in CoSMoS (ACIMS, 2017a).

6.2.1 *Categorizing the Activity Specification*

According to the proposed specification (see Table 6.1), the components in activity-based models can be categorized into composite and primitive. The specifications for the external, internal transition, and output functions are composite. In every external transition function, a bag of external inputs is received and modeled as an activity parameter. The output function sends out the output which is modeled as an activity parameter, although it is an output parameter. The internal transition function can manipulate the state of the model via the use of some primitive components. These abstractions after that can serve as a basis for the behavioral specification of the DEVS atomic model. And their contained components, whether composite or primitive, can collectively form the behavior of each function. Hence, the primitive components can also be used in a different function or possibly different models such as the action of making content or setting the phase to passive. Therefore, the single-action can also collectively form the behavior of the system while maintaining modularity.

The primitive components can also be characterized by their roles in the model, such as changing the state from a DEVS standpoint. Other primitive components may include any action that contributes to the behavior of the model, such as assignment. Primitive components are the fundamental units of the behavior. Their semantics

vary according to their role in the model. In changing the state, the model should primarily determine changes in the phase and the sigma as primary variables for the total state in the atomic model. In some assignment action, the value should assign to its corresponding variable.

The two categories provide a means to establish a stronger context, especially for the primitive components. Since the objective of this work is to provide means for restraining complexity from a behavioral standpoint, the relationship between these components within some context is quite crucial. For example, the activity node takes place within an activity that is created to represent an external transition function. An investigation of whether the same node is simultaneously used in another activity for another element or not could be useful.

6.2.2 Note on Coupled Models and Behavioral Specification

It is important to note that the coupled model has a significant role in delivering the expected outcome of this research. For this reason, at the implementation level, we attempt to put things in a consistent perspective with the existing CoSMoS in which the specification of coupling is supported. CoSMoS recently has been equipped with the capability of behavioral modeling, which is currently based on statecharts as mentioned previously in section 6.1. However, the specification of an atomic model is encapsulated regarding other atomic models. The behavior of a model can only communicate through I/O ports and couplings with other models. Therefore, their abstractions can also differ in a way that preserves this property. An atomic model specified in some form can couple with some other atomic model specified in a different form. In our case, it should be possible to couple models specified in CoSMoS with their behavior modeled using statecharts. Moreover, it should also be possible to couple them with atomic models in which their behavior is modeled using the activity-

based specification as discussed in this work. It is important to help in achieving better outcomes toward the enrichment of the behavioral specification of the parallel DEVS formalism.

Moreover, the mandated behavior via coupling is also crucial in constituting the totality of system dynamics. Since the interaction between different atomic and coupled models is strict via I/O ports and couplings, therefore, their influence on the inner atomic model logic is also restricted to the arrival of the external inputs. Moreover, their impact on other models is also contingent on sending outputs through their output ports.

Many others support the concepts above and capabilities in CoSMoS and integrated with DEVS-Suite. The creation of Parallel DEVS and Cellular Automata is supported in CoSMoS as component-based models (Sarjoughian and Elamvazhuthi, 2009). Then, the corresponding code is automatically generated, although for only the structural aspects. There is no code generation provided for statecharts modeling. This research contributes to providing a richer modeling basis for further code generation and transformations. The path toward having a more disciplined simulation model development requires a collective achievement by multiple abstractions with further emphasize on their complementary role consistently.

6.2.3 Controlled Coupling Using Activities Control Nodes

The provided capabilities in control nodes are used to control the transmission of input/outputs among different atomic and coupled models. Communication of events (i.e., input to input, output to input, and output to output) among all components of any coupled, hierarchial DEVS model are instantaneous. This is described at a high level of system specification to represent the component interactions with each other via I/O ports thereof. The activity abstraction can become useful for

further mandating these interactions among different components. Various control capabilities of the flow can be used to describe the composition of components. The components themselves are modeled separately at a different level. The behavioral specifications can be characterized then by activity specification as we discuss in this work or by various means such as statecharts. In the following, we present two common examples of handling the flow among different components.

Synchronizing inputs

As discussed, multiple inputs can arrive at some input ports and at arbitrary time instances at the DEVS atomic and coupled models. The join node can be used to synchronize various inputs sent among different couplings. For example, the join node can be used to associate two different flows before an adder model. Consequently, the inputs will arrive at the same time, for instance, at the adder input port. Hence, the behavior of synchronizing multiple inputs has separated from the domain model. Therefore, the domain model may or may not account for this need in its behavioral specifications.

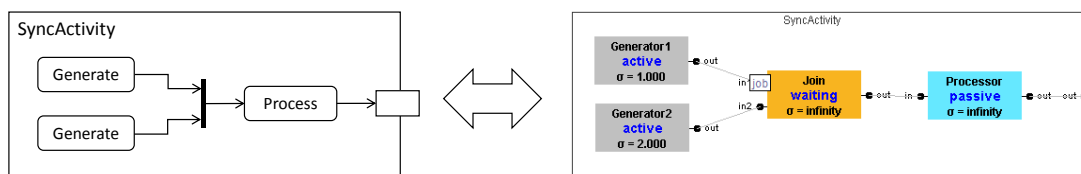


Figure 6.1: An Activity for Synchronizing Outputs from the Generators Prior to Processing. The Simulation View (Right) Is for the Corresponding Implementation in DEVS-Suite. The Join Node as an Atomic Model Is in *Waiting* Phase to Synchronize the Input through the Other Port from the Second Generator.

The join node can be used to specify the synchronization of inputs, as shown in Figure 6.1. Two generators produce inputs for a processor model. The outputs from

both generators are held at some point before their arrival at the processor model. This scenario can be specified as a join node which is modeled as an atomic model with its behavior to be specified to encompass the semantics of the join node. The outputs by generators do not have to produce at the same time instant necessarily. However, having the synchronization ensures the simultaneous arrival of the inputs at the processor's input port. This behavior is defined in the behavior of the corresponding atomic model to the join node, where the model has multiple input ports. Upon the arrival of input, the phase is changed to "*waiting*" to denote the existence of input. Therefore it waits for other inputs from other input ports to arrive the output dispatches upon the arrival of all the inputs. The phase is changed then to *passive* by the internal transition function. The formal specification of the atomic model for the join is defined in Parallel DEVS as

$$JOIN = \langle X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

,

where

$X_M = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y_M = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

$S = phase \times \sigma \times condition1 \times condition2 \times store$, where

$phase = \{“passive”, “waiting”\}$, $\sigma = \mathbb{R}_{0, \infty}^+$,

$condition1 = \{true, false\}$, $condition2 = \{true, false\}$, $store \in X_M$;

$IPorts = \{“in1”, “in2”\}$, where $X_p = V$ (an arbitrary set);

$OPorts = \{“out”\}$, where $Y_p = V$ (an arbitrary set);

$\delta_{int}(phase, \sigma, condition1, condition2, store) =$

$(“passive”, \infty, false, false, x)$ where $x \in X_M$;

$\delta_{ext}((phase, \sigma, condition1, condition2, store), e, X_M) =$

$$\begin{aligned}
& ((\text{"waiting"}, \infty, \text{true}, \text{condition2}, (p_i, v_i)) \\
& \quad \text{if } p_i = in_1 \text{ and condition2} = \text{false} \\
& ((\text{"waiting"}, \infty, \text{condition1}, \text{true}, (p_i, v_i)) \\
& \quad \text{if } p_i = in_2 \text{ and condition1} = \text{false} \\
& ((\text{"waiting"}, 0, \text{condition1}, \text{condition2}, (p_i, v_i)) \\
& \quad \text{if } p_i = in_1 \text{ and condition2} = \text{true} \\
& ((\text{"waiting"}, 0, \text{condition1}, \text{condition2}, (p_i, v_i)) \\
& \quad \text{if } p_i = in_2 \text{ and condition1} = \text{true}; \\
& \delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x); \\
& \lambda(\text{"waiting"}, \sigma, store) = (out, store); \\
& ta(phase, \sigma) = \sigma.
\end{aligned}$$

The atomic model is defined with two input ports and one output port. The model is initially in phase *"passive"* until some input arrives through any port. Two conditions exhibit as secondary state variables to recognize the port through which the input arrives. Both conditions initialize with false. As soon as an input is received, the external transition function toggles the condition corresponding to the input port. For example, if the input is received through port *"in1"*, then the toggled condition is *"condition1"*, and so on. If all conditions are true, the sigma is assigned a zero value causing the output function to occur. The output function then sends out all the stored received inputs. Finally, all state variables set to their initial values in the internal transition function.

Network switch

Just as we can model the synchronization as a join node, we can also model the network switch with decisions for directing the flow of input and output events. The synchronization is not necessary though. The selected control element is the decision

node in terms of activity modeling. The switch directs the flow by sending the received inputs through the corresponding output ports under some defined condition.

As discussed in (Alshareef and Sarjoughian, 2017), the switch model illustrates the handling of multiple inputs arriving simultaneously as in Parallel DEVS. In that model, the switch is characterized to be a controlling component, while the processor model exhibits different behavior. Together, they constitute, along with the other elements, the behavior of the network switch coupled model. The behavior of the processor model is described in detail in section 6.4.

6.3 EMF-based Modeling Engine

It is important to assist modelers in developing specifications for all parts of an atomic model. The proposed approach requires building a modeling engine that supports the activity-based modeling concepts and constructs introduced in Section 6.2. This engine should be robust and powerful enough to handle the potential complexity that might arise in DEVS activity behavior modeling (i.e., from individual to composite parts forming atomic functions). Therefore, we expand our work by constructing a modeling engine that utilizes the previous specifications and translates them into a runtime environment. The resulting engine accounts for formal specifications of DEVS as well as concepts introduced for the atomic model behavioral specification. It also leverages the available tools and technologies to facilitate the process of building those models as well as ensuring the extensibility to allow for future works, including model verification and simulation validation.

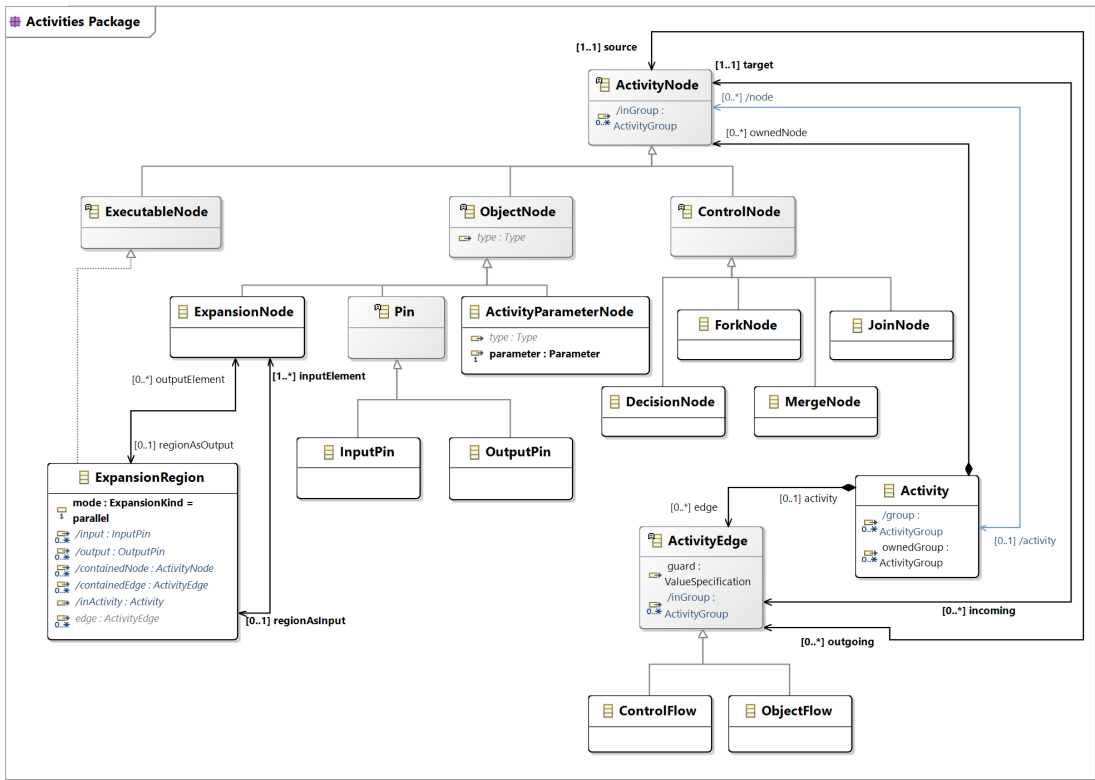
6.3.1 *Activity-based DEVS Ecore*

Using the current realization of the UML2 metamodel in EMF (Eclipse Foundation, 2016a), the Activity metamodel is realized similarly. We try to use it to provide

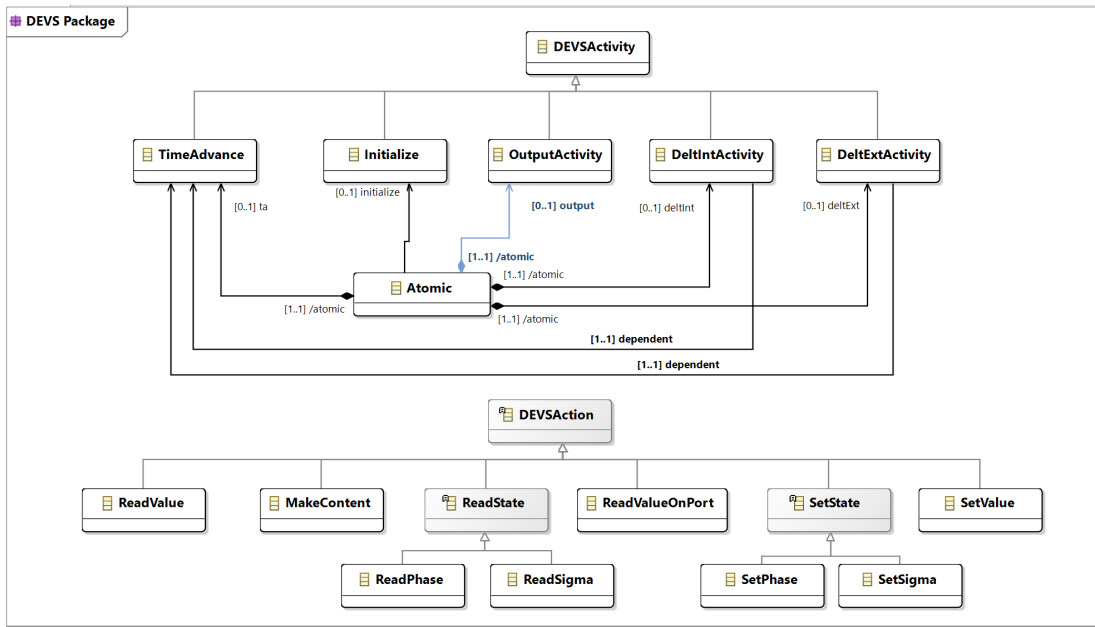
some support for the DEVS atomic model behavioral specification. The created Ecore is currently aimed to be incorporated with EMF-DEVS and used to provide a graphical modeling capability using the Eclipse Graphical Modeling Framework (GMF). The Ecore consists mainly of activities and DEVS packages, as shown in Figure 6.2. DEVS package contains essential elements from EMF-DEVS for both structure and behavior as discussed in previous works (Sarjoughian and Markid, 2012; Sarjoughian *et al.*, 2015) with some additional elements. The activities package contains elements to realize the activity-based specification. They have been created by taking into account the UML 2.5 specification as well as UML2 metamodel (Eclipse Foundation, 2016a) in Ecore.

We construct the relationships between DEVS, activities, and actions metamodels using extensions and references via ESuper types and EReference as defined in Ecore (Steinberg *et al.*, 2008). The state transition functions, output, and time advance functions are specialized from DEVSActivity, that is, specialized from the *Activity* EClass. The goal is to maintain common properties among those four functions in this common abstract DEVSActivity. Thus, they become distinguishable from other activities. References are established in the Atomic EClass for each DEVS function. The atomic model may have up to one of δ_{ext} , δ_{int} , δ_{con} , and λ functions. For example, the reference for the external transition function may not set for the producer model. Therefore, the reference is defined to have zero as a lower bound and one as an upper bound. The references are contained within their corresponding atomic model. It ensures that the strong modularity required for the DEVS formalism is maintained.

Specializing actions for DEVS modeling is accomplished similarly. The behavior can be then characterized by those actions in addition to the comprehensive set of actions defined in UML. The current list of actions as specialized in the Ecore includes (but not limited to) the following:



(a) An Ecore Package Containing Activities Metamodel.



(b) An Ecore Package Containing the Behavioral Constituents of DEVS Metamodel and Actions.

Figure 6.2: Activity-based DEVS Metamodel.

- *SetState*: to change the current state due to some state transition. This component is defined to be abstract.
- *ReadState*: the current state is checked before making changes to some state. This component is defined to be abstract.
- *ReadValueOnPort*: to read input *value* that has arrived via some input *port*.
- *SetSigma*: to assign a time period for some state.
- *MakeContent*: create *(port,value)* pairs where one or more output events are assigned to one or more output ports.

This set of actions is based on the common use in many DEVS atomic models supported in the DEVS-Suite Simulator. However, they are generalized and thus can be employed by other simulators that conform to the DEVS formalism. Their implementations can be specified and mapped according to their definitions in the metamodel.

6.3.2 Activities Graphical Definition and Tooling

The GMF framework supports creating visual editors for modeling languages such as statecharts. We have leveraged this framework and built an expressive graphical notation for modeling behavior based on the activity-based approach. The visual syntax is defined to be consistent with that of the UML activity modeling. Therefore, any activity model created using the graphical activity definition is a legitimate UML activity diagram. The specialized elements for DEVS are given their superclass graphical definitions. For example, *SetPhase* action is defined to have the same visual notation as the one for manipulating the structural feature that it specializes. This

modeling engine supports the development of Activity-based DEVS models such as those shown in Figure 4.4

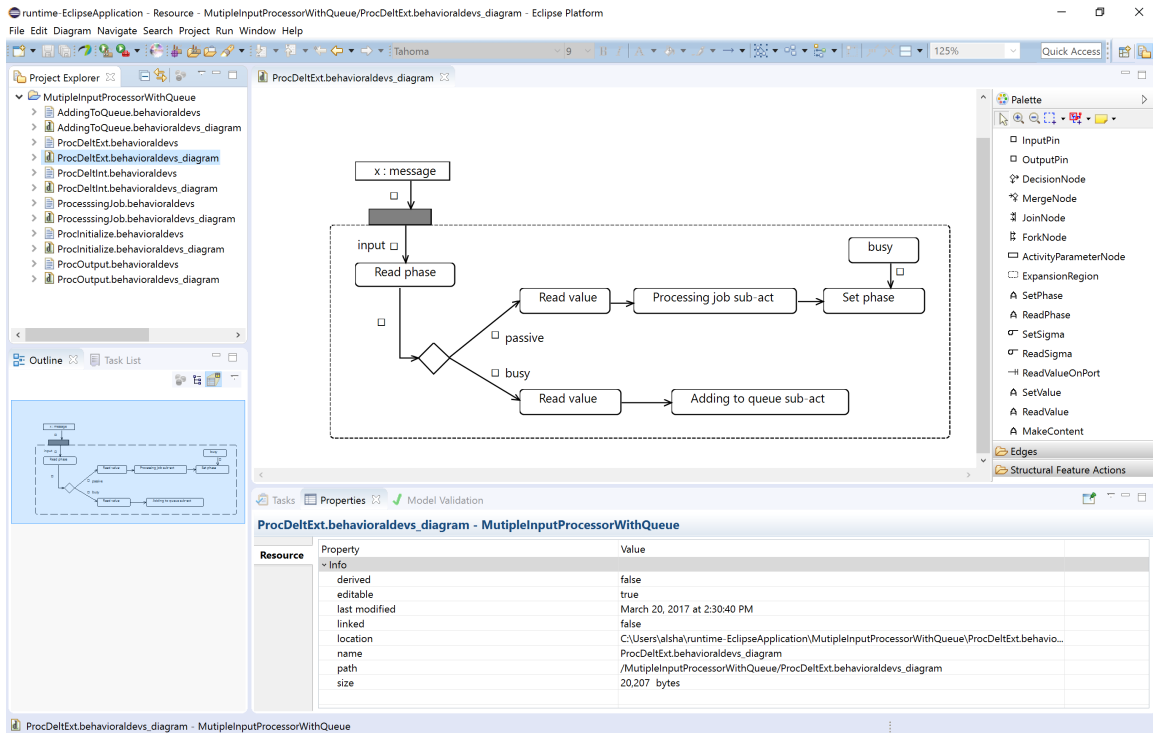


Figure 6.3: Visual Canvas for Activity-based DEVS Modeling.

The canvas for Activity-based DEVS allows creating behavior models through the use of nodes and connectors. The canvas consists of figure galleries, nodes, connections, compartments, and labels. The figure gallery is used to define a set of graphical notations enabling defining the elements of any atomic models as depicted in Figure 6.3. GMF, by default, generates rectangles for node elements and solid polylines for link elements. The definition is modified to represent the graphical notation for activities. The notation includes a variety of unique shapes such as rectangles, rounded rectangles, arrows, and diamond. Customization is often necessary for more specific shapes. A sample of the graphical definition for the decision node is shown in Table 6.2. Graphical definitions for all other elements have been created similarly. Note

that we have made minor changes to some notations for simplicity reasons. One difference is on the expansion node notation by making it a grey rectangle instead of a divided rectangle. The other change, we do not include the rake symbol on the call behavior action. Instead, the action name is appended with *sub-act* keyword at the end.

The tooling definition is created to define the used tools for creating graphical diagrams such as palette and popup menus. Those tools can be then customized further to support specialized graphical interfaces to facilitate the modeling process. Primitive and composite model elements are ensured to satisfy the combined UML Activity and DEVS behavioral model development. This kind of tooling allows for organizing the modeling elements in more useful ways for the modelers and therefore enabling easier and simpler modeling. The UI includes, but not limited to, features like palette grouping, icon images, modeling hints, and guidelines. With these features, modelers are well-positioned to specify complex behaviors iteratively. The palette elements are organized according to their classification from a semantic point of view for activities and DEVS modeling. The palette lists UML activity nodes followed by the DEVS nodes such as the nodes for specifying phase and sigma. Specialized image icons for nodes are provided to capture their mathematical counterparts in the DEVS formalism. For example, *SetSigma* is associated with an image icon of sigma symbol (σ).

6.3.3 Mapping

Finally, we create a mapping model where the elements defined in the metamodel map to their corresponding counterparts in the graphical definitions and tooling. The

| Element | Property | Value |
|--------------------|----------|--------------------|
| Canvas | Name | BehavioralDEVS |
| Figure Gallery | Name | Activities Figures |
| Figure Descriptor | Name | Diamond |
| Rectangle | Name | Diamond |
| | Fill | False |
| | Outline | Flase |
| Polygon | Name | Diamond |
| Template Point | X, Y | 15,0 |
| Template Point | X, Y | 0,15 |
| Template Point | X, Y | 15,30 |
| Template Point | X, Y | 30,15 |
| Node | Name | DecisionNode |
| | Figure | Diamond |
| Default Size Facet | | |
| Dimension | Dx, Dy | 30,30 |
| Node | Name | MergeNode |
| | Figure | Diamond |
| Default Size Facet | | |
| Dimension | Dx, Dy | 30,30 |

Table 6.2: Decision and Merge Node Figure Definition

result leads to combining the three essential models in GMF (i.e., the domain model which is in our case the metamodel, the graphical definition, and the tooling definition). GMF wizard allows distinguishing between nodes and links in the mapping

model. However, it only provides the necessary customization capabilities. The tooling model is manipulated further. The model, graphical, and tooling elements are linked with each other using the node mapping feature and specifying its properties accordingly.

6.3.4 Preliminaries on the Validation of the Activity-based DEVS Models

The validation of the proposed approach is currently based on the provided validation capabilities by EMF. The metamodel is shown in Figure 6.2 captures the concepts by representing them in a set of classes, attributes, and references. It after that forms the ground for further constituting the validity of the instantiated models. After being specified, the validation is performed by the EMF engine.

Although a thorough validation study is considered as future work, we briefly discuss some of the validation capabilities and benefits obtained for the behavioral specification. Many of these benefits are acquired through the merit of the existing metamodel itself in general. Also, many others are also acquired by virtue of effectively utilizing the underlying framework. EMF engine has a validation engine that allows defining and checking the validity of model constructs. Any validation can be conducted by merely customizing the default properties of different models as far as going all the way to modifying the generated code and defining more constraints yields possibly to a more restrictive modeling environment. Therefore, the legitimacy of the constraint must be ensured to be valid for all corresponding models before enforcement. For example, the value of the sigma must always be non-negative. This constraint is defined and enforced with an error message in case of violation. Less restrictive constraints can also be specified. In the case of not receiving any inputs, a warning message shows up. Some particular circumstances in the model do not have to receive inputs such as an autonomous generator model (i.e., has no input ports).

For any non-mentioned validation, using EMF-based metamodel provides a suitable ground nonetheless to simplify the process of adding further analyses by defining constraints or through extension mechanism. Such validation can be for a general property or specific to a particular domain. This can serve for the behavioral specification of domain-specific models (e.g., (Zhu *et al.*, 2017)). A framework is proposed to deal in part with complications that are likely to arise during model development regarding behavior. Linking to such domains can be possible after establishing the necessary extensions and transformations.

Although some constraints can be validated to ensure the legitimacy of model specification, validation in the sense of simulation is impractical due to the infinite state space of DEVS models. Further constraints can be made for a specific domain on the model inputs, outputs, and states, for validation purposes. The constraints for models complement the validation of simulated behavior. They can apply to the state transitions by controlling the elements that deal with state changes (e.g., order in which state or non-state variables can change their values). Extracting inputs can also be checked in the reading inputs and ports elements in conjunction with their corresponding sets as defined by the model. The same can also work for outputs. Such ability has been possible after dissecting the behavioral modeling approaches. Processes such as identifying, categorizing, and characterizing each element, help to look into more specifics after that.

6.4 Activity-based Modeling for Multiple Input Processor with Queue

A model of a processor with a buffer exemplifies the essence of the Parallel DEVS formalism. The processor can handle multiple input events (e.g., jobs to be processed) from the standpoint of their arrival and storage. Various inputs may arrive simultaneously on one or more input ports. The buffer is used to store jobs when

the processor is busy processing another job. The saved jobs proceed in the order in which they have stored in the queue. The model is defined in Parallel DEVS (Zeigler *et al.*, 2000) as

$$Processor_{queue} = \langle X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

,

where

$$IPorts = \{\text{"in"}\}, \text{ where } X_{in} = V_X \text{ (an arbitrary set);}$$

$$X_M = \{(p, v) | p \in IPorts, v \in X_{in}\} \text{ is the set of input ports and values;}$$

$$S = \{\text{"passive"}, \text{"busy"}\} \times \mathbb{R}_0^{+\infty} \times q;$$

$$OPorts = \{\text{"out"}\}, \text{ where } Y_{out} = V_Y \text{ (an arbitrary set);}$$

$$Y_M = \{(p, v) | p \in OPorts, v \in Y_{out}\} \text{ is the set of output ports and values;}$$

$$\delta_{int}(phase, \sigma, q)$$

$$= (\text{"passive"}, \infty, q) \text{ if queue is empty}$$

$$= (\text{"busy"}, processingTime, q')$$

otherwise remove head of the queue;

$$\delta_{ext}((phase, \sigma, q), e, ((\text{"in"}, x_1), (\text{"in"}, x_2), \dots, (\text{"in"}, x_n))), \quad x_i \in X_{in}$$

$$= ((\text{"busy"}, processingTime), x_1, x_2, \dots, x_n) \text{ if phase = "passive"}$$

$$= ((phase, \sigma - e), q.(x_1, x_2, \dots, x_n))$$

otherwise add input events to the tail of the queue;

$$\delta_{con}((s, ta(s)), x) = \delta_{ext}(\delta_{int}(s), 0, x);$$

$$\lambda(phase, \sigma, q) = (\text{"out"}, q.head), \quad q.head \in Y_{out}, \text{ if phase = "busy"};$$

$$ta(phase, \sigma, q) = \sigma.$$

Activity models are created for the internal transition, external transition, and output functions (see Figure 6.4). Specifying the remaining function is straightforward.

ward from an activity modeling standpoint. Modeling some parts of the processor model (e.g., queue) is currently out of our scope. However, they are being manipulated in which their impact cascades to the specified behavior. We note that the initialization and termination of the modeled activities do not appear. Their exclusion is due to the simulation protocol being responsible for handling the execution order of different functions in a way that complies with the operational semantics of the abstract simulator.

A notable advantage of the developed modeling engine is that it enables the modelers to visualize a more comprehensive array of DEVS activity models. It allows viewing the model in various ways, including textual representation, although this representation was not our primary focus. The powerful unifying capability attained by the reliance on Ecore reduces some difficulties associated with modeling approaches, including the UML state machine.

6.4.1 Interpreting the Processor Model in the DEVS-Suite Simulator

The path toward concrete realizations of the proposed abstraction can take place through model transformation. More elements have to be accounted for since the concrete models are more restrictive than their representations at some higher level. Concepts such as visibility and data typing need to be strictly defined by the transformation to achieve fully specified models. In the case of the DEVS-Suite simulator, the code that corresponds to the atomic model can be obtained via interpretation or code generation. The manipulation of the primary state variables, which are the phase and sigma, is more accessible than the secondary ones, especially with specific parts such as a queue. The queue is classified as a non-simulatable model (Sarjoughian and Elamvazhuthi, 2009) as compared with the simulatable processor model. Therefore, since our focus is the behavioral specification, we assume that the queue is already

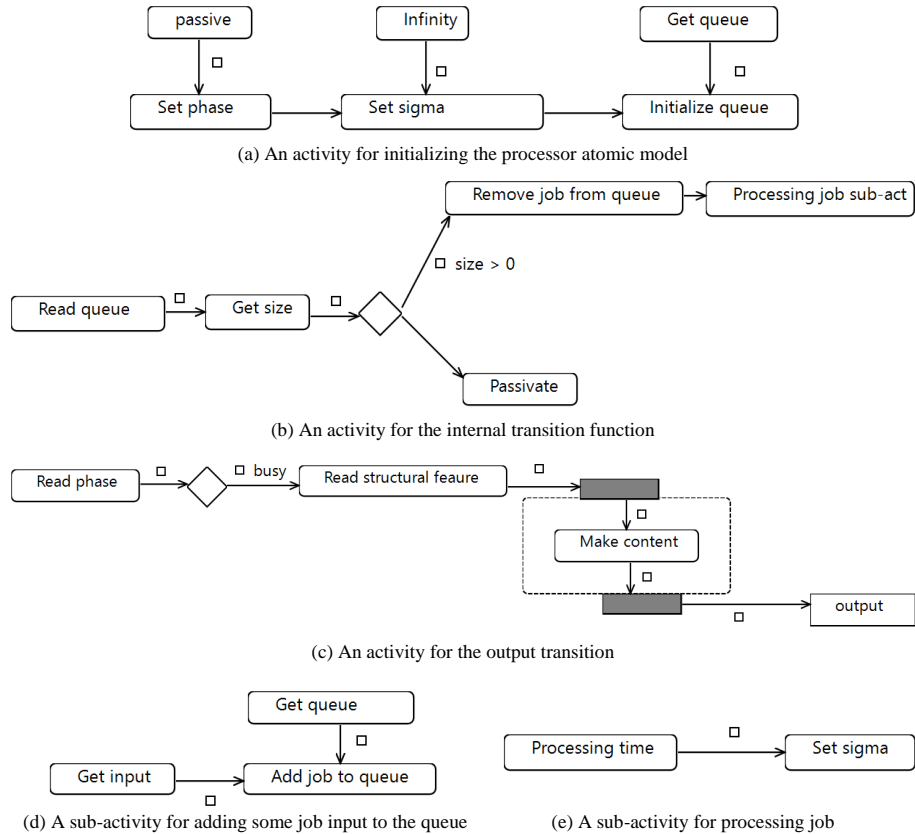


Figure 6.4: Activity-based DEVS Modeling for Multiple Input Processor with Queue.

defined as in (Sarjoughian *et al.*, 2015). As a result, the manipulation of its elements could be performed directly during the transformation. The implementation of the external transition function shows in Listing 6.1. The *for* loop is for checking any bag of received inputs. Then, the phase is checked as in the *if* statement in correspondence to the decision node in the activity model. The phase is defined as a *String* in the DEVS-Suite simulator. We consider that in the transformation, not in the activity model. The sigma is also manipulated similarly, while the data type is *double*. The details are accounted for during the transformation to complement models at a higher level with the necessary constructs and definitions to make them concrete.

Listing 6.1: External transition function for the processor with queue

```
public void deltext(double e, message x)
{
    Continue(e);

    for (int i=0; i< x.getLength(); i++){
        if (phaseIs("passive"))
            for (String inPort : inPorts)
                if (messageOnPort(x, inPort, i))
                    {
                        job = x.getValOnPort(inPort, i);
                        sigma = INFINITY;
                        phase = "busy";
                    }
            else if (phaseIs("busy"))
                for (String inPort : inPorts)
                    if (messageOnPort(x, inPort, i))
                        {
                            job = x.getValOnPort(inPort, i);
                            q.add(job);
                        }
    }
}
```

6.5 Conclusion

The use of various modeling approaches and methodologies for behavioral modeling is invaluable as it paves the way for reasoning about the system's properties and dynamics. We aim to contribute to this need by providing a basis for this emerging approach for the behavioral specification of discrete system modeling and specifically for the parallel DEVS formalism. We described the basis for Activity-based DEVS modeling. We considered different views for adopting activities as a candidate behavioral modeling language for specifying DEVS atomic model (and by extension coupled) behavior. The view has to account for compliance with concepts from formal and semi-formal modeling methodologies to enable and enrich the utilization and enrichment of model development. We proposed an approach where behavioral abstractions in the DEVS formalism and statecharts are extended with those of the UML activity.

We chose MDA as the basis for defining foundational artifacts key for detailing behaviors belonging to the DEVS atomic model functions. The potential value of behavioral metamodeling concepts for developing simulation models can be unlocked by proper employment to that among different MDA layers. As the complexity and scale of systems continue to grow, disciplined use of the DEVS modeling formalism and the model-driven development, especially concerning system behavior simulation, is attractive.

We demonstrated the approach by developing a first-generation Activity-based DEVS modeling engine supported with multiple capabilities using the GMF framework and tool. This engine facilitates developing atomic model behaviors at scale. This GMF-based tool supports evolving model behavior with automation (e.g., ensuring conformance to the DEVS syntax and semantics for functions).

Promising future work includes furthering behavioral artifacts supporting higher-order control structures. These can be key for building more useful simulations for Systems of Systems including Cyber-Physical Systems and Internet-of-Things. Another future research interest is to reduce the size of behavioral models similar to design and code refactoring.

PARALLELISM SEMANTICS IN MODELING ACTIVITIES

Abstractions are indispensable for human understanding as a way of thinking about a complex world. Models are representations of the world, and yet themselves may become quite complex, each of which is some structure or behavior abstraction. Having a way of abstraction is fundamental in constructing models, yet poses key challenges. An effort to establish a stronger grounding for modeling methodologies and frameworks resulting in new insight is highly beneficial. A possible path can be sought through approaches for incrementally constructing complementary models at multiple abstraction layers.

Abstractions for systems are created using informal, semi-formal, and formal methods. The latter method can be captured conceptually in some sets of structural and behavioral specifications (see Figure 7.1). Different combinations from multiple categories can be selected to serve as a framework in which numerous formalisms, languages, and methods can be used. For example, a collection of modeling methods such as the UML Activities, State Machines, and Class can provide powerful computational abstractions. When adhering to strong metamodeling concepts, the development of these semi-formal methods as well as formal methods can be aligned to benefit from their potential synergy collectively.

Complexity issues may arise within models with relatively non-trivial structures and behaviors. It is essential to account for that by restraining the complexity and scale traits of models (Sarjoughian, 2017). For semi-formal methods, the encountered complexity can also associate with a high level of ambiguity, which can make the problem even more difficult. Any formalization effort could be subtle since it may

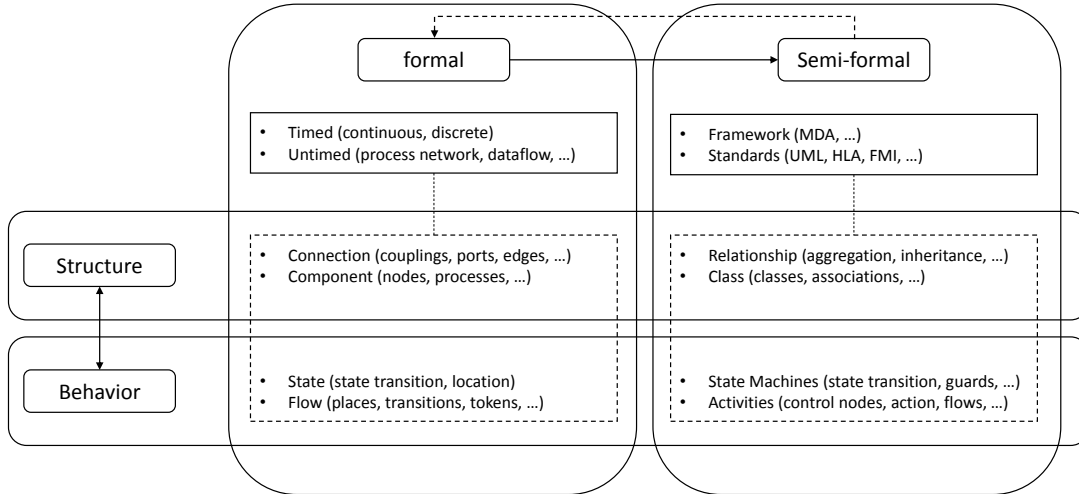


Figure 7.1: A Classification for Formal and Semi-Formal Component-Based Modeling Approaches with Respect to Structure and Behavior.

result in sacrificing some general concepts and specification expressiveness.

Through enriching abstractions (e.g., Activity diagrams), we do not only benefit by understanding them in multiple ways but also making their use more straightforward given different structural and behavioral specification granularity. The challenge of defining constructs and artifacts for behavior is demanding, in part due to time-based, parallel properties intrinsic to dynamical systems. Therefore, there is quite a need for efforts that may help in the further restraining of complexity and scale associated with dynamic behavior.

Introducing the concept of action can be useful for component-based modeling to allow modelers to delve into behaviors that have a relatively high level of logical and time complexities. In the DEVS formalism, the concept of action is absent in general (except in RT-DEVS (Cho and Kim, 1998) and ALRT-DEVS (Sarjoughian and Gholami, 2015) where the context is real-time modeling). We think that action can provide a suitable means to complement the state-based behavioral specification of Parallel DEVS models. Thus, we consider the action as a bridge for establishing

abstractions in general, and for multi-processor architectures specifically by which we use throughout the paper to exemplify the discussed concepts. The abstraction is proposed to complement both the DEVS formalism and other action-based behavioral modeling approaches with the necessary definitions to overcome challenges associated with behavioral modeling.

Although it is quite advantageous (or even necessary) to increase the level of expressiveness, especially with behavioral specifications, it could be the case where the semantics become obscured. In this work, we consider the parallelism semantics as a way to investigate the activity modeling approach, as presented in the UML 2.5 (OMG, 2012). Behaviors are observed through the lens of the DEVS formalism (Zeigler *et al.*, 2000), along with its extension Parallel DEVS (Chow, 1996) by which parallelism is further exploited among different state transitions and the handling of events. In doing so, we are not aiming a new formalization of the UML activity, but rather dissecting into the semantics of its essential constructs, in particular, control nodes. The modeling and simulation of UML activities in Parallel DEVS have been proposed in previous work (Alshareef and Sarjoughian, 2017).

This paper is organized as follows. We discuss the role of action in behavioral modeling, especially in activities and for modeling in DEVS in Section 7.1. Then, we discuss some architectures for parallel processing in DEVS along with their proposed abstractions in Section 7.2. We examine parallelism semantics with control constructs in activities in Section 7.3. Then, we discuss related works in Section 7.4. Finally, in Section 7.5, we summarize this research with a discussion on future work.

7.1 The Role of Action in Activities and Other Behavioral Models

An action is a fundamental unit of behavioral models. A set of actions partially describes the situation in which the behavioral model as a whole is specified. Actions

are linked in certain ways or associated with other artifacts per a chosen modeling language. The order or the control mechanism, by which actions connect together, constitute the overall behavior. Some approaches allow explicit specifications of the actions along with control, such as in the activity modeling. In general, each specification elaborates through introducing new or enriching existing elements in support of describing non-trivial behaviors. Complexity in a behavior arises quickly and quite overwhelmingly. Therefore, a potential role of actions is further restraining the complexity in systems where action-level specifications are realized.

In UML 2.5, control nodes are major subordinates in modeling activities. The node can be either *initial*, *final*, *split*, *join*, *decision*, or *merge*. The *initial* and *final* are used to initiate and terminate an activity, respectively. We exclude them from the focus of this work. The *split* and *join* nodes are visualized with an opaque rectangle while the *decision* and *merge* nodes are visualized with a diamond. The *split* node receives one incoming flow and produces multiple concurrent flows. The *join* node receives multiple flows and produces only one. The flow can be a control or data.

Further knowledge about flows is necessary for handling individual cases where multiple flows for a single node vary. The *decision* may take up to two incoming flows and produces multiple. Each outgoing flow is associated with the condition that evaluates to determine which flow should be selected to direct the incoming flow. The *merge* node receives multiple flows and produces only one. As opposed to *join* node, it addresses the flow as soon as it gets one of the incoming flow without waiting for others. The focus of this paper is on these four nodes and their abstraction roles in representing the archetype architectures for multiple processing units.

7.1.1 Atomic Model and Action

The formal specification of the parallel atomic model

$$DEVS = \langle X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle \quad (7.1)$$

is defined to provide the basis for constructing discrete event models in hierarchical and modular form. Based on modularity, the system can grow in a well-formed manner via coupling and composability. By understanding Classic DEVS, Parallel DEVS, and relations between them, modelers are obliged to ask subtler questions about models and their semantics. Thus, we propose DEVS specification for creating a correspondent atomic model for action (Alshareef, 2017) to be able to understand their encompassing behavioral models based on system-theoretic concepts. The processor model is used to describe the abstraction of action. Other atomic models are used to describe the abstraction of the other control elements. Together, they formulate a chain of actions (i.e., coupled models) (Sarjoughian, 2017). A wide range of execution semantics can be examined based on a time base and when generating time trajectories for observing behavior. Time advance, for instance, and processing time, in particular, is well suited to capture semantics that is relevant to duration with actions.

The processor as an atomic model is well described in (Zeigler *et al.*, 2000). It can be in either a busy or passive phase. The jobs arrive through one or possibly multiple input ports. While not receiving any external input event, the processor remains in a passive phase indefinitely. Upon receiving external inputs, the processor transitions to an active phase and may store the job in unitary or multiple unit storage such as a queue. The processor produces output or multiple outputs also through one or multiple ports simultaneously. As far as the action is concerned, it can take place within a context of a behavior classifier. Action in this context can be associated with some input and output pins, which may cause changes in the state of the model according to some semantics.

7.1.2 *State and Time for Actions*

Introducing the concept of the state allows us to enjoy the progress and profoundness made in the theory of modeling and simulation. The notion of action has been evolving throughout the history of computing. In (Gelfond and Lifschitz, 1993), action, or a series of actions, is used to describe the world in addition to state transitions. In (Shoham, 1989), the author identifies the concept of action is primitive as a problem because of isolating time. Then, he proposes a time-aware framework to extend action as defined in some AI systems with the notion of time. Understanding system dynamics has to be based upon temporal structure to enable describing the world after that. Analyses about the deadline, delay, and concurrency can ensue. In this framework, actions define states, and they are not merely connecting or serving as intermediaries between states. That is, an action in some given state defines the next state. In parallel DEVS formalism, the next state is defined by the state transition functions. For example, in the external transition function, the next state is defined by the current state, inputs, and elapsed time.

The notion of time is inherent, but not the action. The concept of action is introduced within some variants of DEVS, namely Real-Time (RT) DEVS (Cho and Kim, 1998) and Action-Level Real-Time (ALRT) DEVS (Sarjoughian and Gholami, 2015). The latter has been proposed to support defining real-time constraints at the action level from both modeling and simulation point of view alongside the concept of locations defined for real-time statecharts (Giese and Burmester, 2003). That is, the modeler will be able to develop a real-time model. Under certain conformance conditions, the real-time simulator (DEVS-Suite (ACIMS, 2017b; Sarjoughian and Gholami, 2015)) will be able to simulate this model in which the abstract simulator has been extended for real-time systems. It fundamentally depends on parallel and

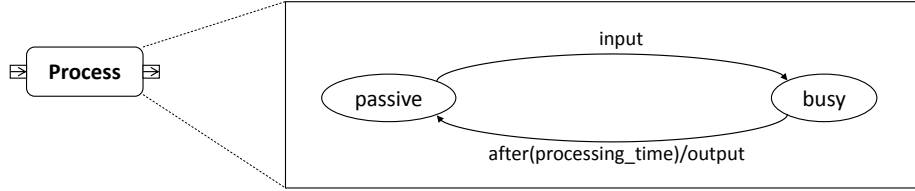


Figure 7.2: A View of an Action and State.

real-time DEVS and real-time statecharts.

By introducing the concept of state (see Figure 7.2), behavioral models are equipped with actions by which both conceptual and design benefits can be obtained. Therefore, we can conduct further analyses and probably reason about actions. Modelers may or may not assume that actions are instantaneous, or they happen with a zero time advance. The general formalism does not impose restrictions in this regard, although modelers ought to decide on which restriction has to take place to attain some benefits such as verification. Action as an atomic model is inclined to the ordinary minimum requirement from a system-theoretic standpoint for the model to be non-trivial (Wymore, 1993). Two state variables are defined for the action. The first variable is *phase*, where at least two values can be given for the model to be in. The second variable is *sigma*, where time duration for a given phase is assigned zero, a positive real number, or infinity value.

7.2 Multiprocessor Architectures

Different architectures for computational processors have been extensively discussed in the literature from DEVS viewpoint (Sarjoughian and Zeigler, 1998). They deliver different outcomes at various computational efficiencies (e.g., turnaround time and throughput). It is essential to have a means for investigating these architectures to help modelers to understand and conduct studies about them. For example, properties such as throughput and turnaround time are essential for making key decisions

for processes in different domains and technologies such as web-services. With Parallel DEVS, evaluations of different architectures can take place by examining their performance measurements either individually and relative to each other. Other benefits can be attained by the employment of other variants of the DEVS formalism such as the Finite and Deterministic DEVS (Hwang and Zeigler, 2009) and Constrained-DEVS (Gholami and Sarjoughian, 2017). Abstractions become important to make these benefits accessible in different domains while grounding them with a fundamental theoretical basis. Architectures such as multi-server can reveal valuable insights when examined in simulation and model checking contexts.

In the simplest case, jobs can be assigned and processed by a single processor. Other cases, where the architecture includes multiple processors, involve a coordinating procedure to assign jobs to their designated processors and handling them under pipeline or divide and conquer regimes. In general, jobs get assigned to processors according to the concept of partitioning, which is, to some degree, the basis of parallel computing (Wilkinson and Allen, 2005). The general goal of these multi-processor architectures is to process externally received jobs as fast as possible with as few resources as possible. However, regarding performance, further elaboration can be made.

The performance of the architectures can evaluate properties such as processing time after setting some objective. A common goal would be to increase the efficiency of the architecture by keeping processors busy for as long as possible without losing any jobs. Coordinators can manage the distribution of jobs among processors. Moreover, the performance of the architecture according to the set objective, is examined through an experimental frame (Rozenblit, 1991). In our approach, the processor as an atomic model is realized in correspondence with the activity modeling approach as an abstraction (Alshareef and Sarjoughian, 2017) to capture the concept of action.

Then, after introducing the concepts of time and state, we create our models along with their corresponding abstractions to come up with the necessary observations regarding the semantics of parallelism. In Figure 7.3, we show archetype architectures for processing jobs along with their possible counterpart abstraction for activities.

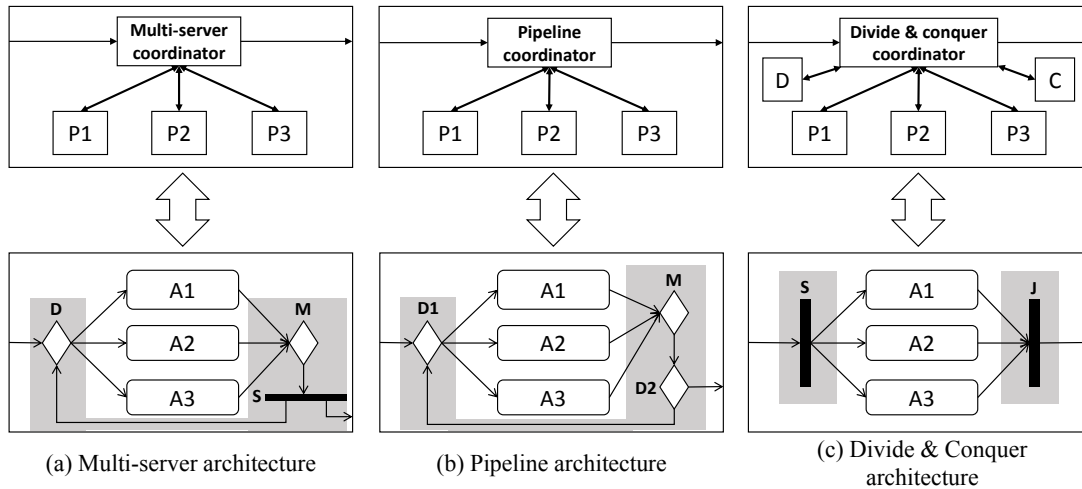


Figure 7.3: We Examine Different Abstractions for Different Architectures. The Component Views with Couplings Are Shown in the Top, and Their Corresponding Activities Are Shown in the Bottom. The Areas in Grey Highlight the Control Nodes That Are Used to Represent the Coordinating Procedure in Each Architecture. The Letter P Stands for Processor and A for Action. In (A), the Coordinator Is Represented by the *Decision* Node D to Direct the Job According to Some Condition Associated with the Outgoing Flow. Conditions Are Visually Omitted. In (B), the Job Is Either Brought Back to the *Decision* Node d1 to Be Directed Again, or Sent out If Completed. In (C), the Job Is Divided in the *Split* Node S and Combined Back in the *Join* Node J after Processing.

7.3 Parallelism Semantics

Parallelism is exploited to some extent within the context of UML. We focus on activities in which parallelism is exploited using the semantics of specific control constructs. In Table 7.1, we show a brief explanation of the syntax and semantics along with a brief discussion on how to handle such semantics in DEVS. In the following subsections, we intend to elaborate further on some elements with details relative to their role in parallelism. The observations are made visible within the context of modeling and simulation. While having abstractions is certainly useful for facilitating modelers' understandings, enriching them is also beneficial from both conceptual and concrete standpoints. In activities, the overall behavior is broken down into a set of activity nodes where action comes into play (along with other elements such as control nodes) as a specialization of the activity node. It is necessary to examine semantics of individual elements to deduce about their overall behavior as a collective.

As shown in Table 7.1, the semantics of *Join* constitute the synchronization of multiple incoming flows with the possibility of collisions between internal and external transitions and the simultaneous arrival of inputs. From a DEVS perspective, a parallel atomic model with multiple input ports can be specified with the capability of processing multiple inputs. The confluence function is responsible for resolving order between state transitions. The elapsed time can be used for any state interruption, although according to (OMG, 2012), interruption is not allowed in some specific scenarios when executing the *Join* node.

Without a temporal structure, it is hard to draw observations about the *split* node, for example, and its subsequent concurrent flows whether they are control or data flows. In general, the ordering of the action execution, which comes at the heart

Table 7.1: A Subset of Activities Elements and Briefly Their Semantics with Respect to Parallelism in Correspondence with DEVS

| Syntax | Semantics | DEVS |
|------------------|--|--|
| Join | Synchronizing multiple flows and multiple inputs | Atomic model with multiple input ports with confluence function for output collision |
| Split | Concurrent outgoing flows | Atomic model with multiple output ports with confluence function for input collision |
| Action | Actions can be within concurrent paths | Atomic model with some input and output ports and may execute concurrently with another components |
| Expansion region | Multiple input processed in iterative or parallel manner | Coupled model with an atomic model receiving a bag of inputs |

of activities as opposed to state machines, may not hold (see Figure 7.3c). We note that it does not necessarily mean that the multiple paths after the *split* node must execute in parallel. Moreover, it also does not have to enforce an order for executing them. That is, the execution should also be valid if it is accomplished in a parallel manner.

In the view of representing an action as an atomic model, the modeler can account for these details and more due to the high level of expressiveness encountered in Parallel DEVS. The component-based simulation accounts for modularity and hierarchy. Two components can only communicate through ports, and their influence on each

other shall be accounted for in the modeling process. Based on the simulation time, the component may proceed in parallel according to the model specifications without implicit enforcement of some order. Lifting such constraints can be beneficial for modelers since it allows them to manipulate behaviors of the model in a more dynamic yet well-formulated way.

7.3.1 Multiple Branching via Split Nodes

Split (also called fork) node allows for controlling the flow by splitting the incoming flow into multiple outgoing concurrent flows. Activities in UML do not lend themselves to DEVS or any other formalism with explicit time notion or with stronger support for atomic operation (i.e., Lamport, 1986). However, the semantics of split nodes are inclined to further elaboration concerning time. For example, the subsequent nodes of the *split* node shall receive the produced flows at some point in time, which may vary. A failure in receiving the flow by the subsequent node at the time of producing it does not necessarily mean that the flow is lost. It instead indicates that the target may accept the flow at a later time. Other issues can arise during processing the split node and the nodes that are after it (possibly actions). As mentioned earlier, some notion of time has to exist to attain a valid execution of the model.

In Parallel DEVS, multiple outputs can dispatch through different output ports simultaneously, which makes it possible to capture the semantics of the split node for its outgoing flows. Therefore, outputs are guaranteed to arrive at their corresponding target nodes simultaneously too. However, they are not necessarily accepted or processed. This can be due to different reasons among which the state of the receiving node or action, whether it is passive or busy. It may also differ according to the capability of the node to store data. Similar to that, the processor model can be with or without a queue. On the one hand, the processor without queue may lose

jobs when it receives them while in a busy phase processing other jobs. On the other hand, the processor with a queue starts processing the jobs when it receives them in phase idle. It only stores them in the queue for later processing when the jobs are received while in phase busy. Another aspect is the concurrent execution of actions (see Figure 7.3c, action A1 and A2). These actions are enabled after the split node. Then, they may (or may not) proceed at the same time. The formalism is expressive to capture the different scenarios that might arise. The ordering, in this case, is not strictly imposed, neither it should. The execution of the nodes may take the order A1 then A2, A2 then A1, or they can proceed in parallel. None of these orders shall be imposed according to the semantics (see Figure 7.4c for the behavior of a possible scenario). The formal specification of a correspondent atomic model to the *split* with two output ports is the following

$$SPLIT = \langle X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

, where

$X^b = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y^b = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

$S = phase \times \sigma \times task$, where

$phase = \{passive, sending\}$, $\sigma = \mathbb{R}_{0, \infty}^+$, $task \subseteq X_M$;

$IPorts = \{in\}$, where $X_p = V(\text{an arbitrary set})$;

$OPorts = \{out1, out2\}$, where $Y_p = V(\text{an arbitrary set})$;

$\delta_{int}(phase, \sigma, task) = (passive, \infty, task)$;

$\delta_{ext}((phase, \sigma, task), e, X^b) = (sending, 0, (in, v_1), \dots, (in, v_n))$;

$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x)$;

$\lambda(sending, \sigma, task) = ((out1, task'), (out2, task))$;

$ta(phase, \sigma) = \sigma$.

We note that the outgoing flows from a *split* node are synchronized. Synchronizing can take place by sending the same output (duplicates) via the same port through different couplings, or via different output ports through different couplings simultaneously. In the case of data flows, it is not necessary to send out the same data, although possible. Such need explains the rationale to use two different output ports to allow different outputs to take place. If the same output is being duplicated and sent out to different destinations through different couplings, then one output port suffices. However, the formalism lacks a mechanism to identify duplicates. Therefore, there has to be some handling for this issue in the model if needed. In a case of multi-processor, the *split* node sends *task'* through the output port *out1* to notify the destination that the task has been processed while it sends *task* after processing

through the output port *out2*. It allows for distinguishing between the arrival of new tasks and the notification of task completion.

7.3.2 *Joining Multiple Paths and Interruptions*

Join node is used to combine multiple incoming flows. The node waits for flows whether they are control or data to arrive before producing the outgoing flow. According to UML, if the *join* node is associated with specifications, then it must not be interrupted by any incoming flow during the evaluation process. From a modeling perspective, we think this is quite restrictive. Unless the execution is assumed to be instantaneous, the modelers may need means to account for such interruptions, which can occur via elapsed time. Also, the modeler might need other means to account for combining the multiple incoming inputs.

Therefore, the correspondent parallel atomic model shall account for such an elaboration in the following means. Firstly, multiple inputs may arrive simultaneously within the same or different input ports. Secondly, an interruption might happen during the state, especially while waiting for inputs from other input ports to arrive. Thirdly, a combination mechanism of the different inputs shall take place before producing the outputs. For instance, identical inputs can merge into one output.

We note that a queue is defined in the *join* node to handle the arrival of inputs through the same port before the arrival of input from the other port. Since the node has to wait for input through each incoming flow, the input may interrupt the node while it is waiting for the input from the other port. In this case, the input will be stored in a FIFO queue and popped out upon the arrival of the input that corresponds to it from the other port. In the case of collision among transitions, the internal transition function executes first. The formal specifications of *decision* and *join* nodes are discussed in (Alshareef and Sarjoughian, 2017) and (Alshareef *et al.*,

2018) respectively.

7.3.3 Using Control Nodes for Job Coordination

The multi-processing architecture is usually equipped with a coordination unit for scheduling and distributing jobs among the corresponding processors. In our case, the coordinating unit is represented by an atomic model (or a set of atomic models) that captures the semantics of the *split*, *decision*, *join*, *merge*, or some semantics for other control element. An atomic model (or possibly multiple) is specified for each. We demonstrate three different types of architectures, a coordinator that assigns jobs to multiple processors, a pipeline, and another with a divide and conquer coordinator. In all cases, the processors may execute concurrently, but not necessarily. Based on the conducted experiment, the coordinator determines the assignment or the partitioning of tasks among one or different processors. Then, the actions of processing the jobs may take place concurrently according to the logical time by the simulator. Figure 7.4 shows the component view for the multi-processor architecture. It also shows some time trajectories for each architecture.

In architectures with multiple processors, the coordinator is responsible for the assignment task under different conditions. Any outgoing flow from the decision node is associated with a condition. The decision is used to model the condition under which the flow will direct to a specific target action with the possibility of directing multiple flows to different target actions. However, each output will only direct to one action. We use this abstraction for the assignment of jobs for different processors. While the coordinator eventually collects jobs, this is accounted for in the activity abstraction by merging node. The node redirects the flow as soon as it receives anything through one incoming flow. After that, the split component sends outputs to the decision component to notify it about the job completion and through

the external output coupling. The component view in Figure 7.4a corresponds to the architecture and abstraction shown in Figure 7.3a. The I/O trajectories are produced for the scenario of injecting a job at the beginning of the simulation (at time 0, see the left side in Figure 7.4b). On the right side of Figure 7.4b, the phase, and sigma superdense time trajectories are produced to demonstrate the sending transitory state (since it has a zero time advance).

In divide and conquer architecture, the job can divide into multiple jobs for many reasons among which delivering better utilization of processors or making the job simpler to process by an individual processor. The procedure starts by dividing the job according to some mechanism, then conquering each part, and ends up combining them concerning the dividing mechanism. The semantics are akin to their counterparts in both *split* and *join* nodes for dividing and combining respectively. The semantics of action among the concurrent paths are partially captured in the processor. This architecture is suitable for a wide range of problems. The parallelism resides in many aspects during processing at the atomic and the coupled level. The outputs of the coordinator are dispatched concurrently with a strong simultaneity. Then, they are also received at the same time instant by their corresponding processors. Although not necessary, the processors may execute concurrently. However, their outputs for the processing cycle shall produce before the combining procedure.

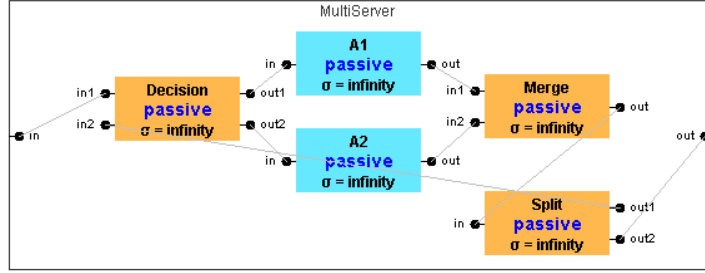
Regarding the pipeline architecture, it is possible to have a series of processing components in a simple pipeline. However, the created abstraction is made by modeling the coordination unit in a set of decisions and merge nodes. The first decision is made to direct the job to the corresponding node. The other one is to decide whether the job has completed. If yes, then it gets directed to the external output port. If not, then it gets directed back to the first decision node. Multiple jobs may arrive simultaneously through the same or different channels with also the possibility of

collision between external and internal transitions. The architectures differ mainly in the condition for assigning jobs and the control nodes. In Figure 7.4, one animation view of multi-server architecture is shown. We plan to discuss and show others with more details in future work. A scenario takes place in such a way; a job injects upon the beginning of the simulation run of each coupled model. In the first cycle, the job (perhaps multiple jobs as in divide and conquer) directs to the right component. Then, the time trajectories for the phase variable are observed for each, as shown in Figure 7.4c until the processing completes. The trajectories are produced using the component tracker feature in DEVS-Suite 3.0.0 (ACIMS, 2017b).

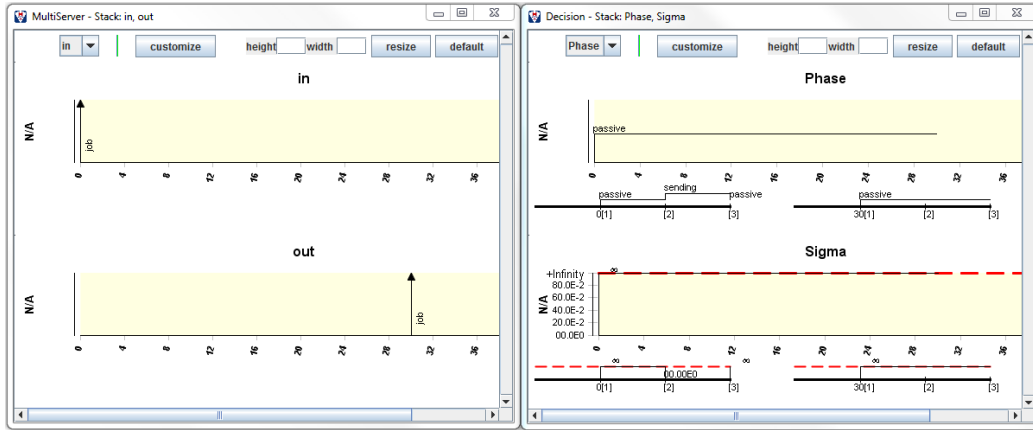
7.4 Related Work

We consider works that focus on the importance of behavioral specifications in the context of modeling and simulation. The recently adopted foundational UML subset (fUML (OMG, 2013)) has been developed to support executing models. As a semi-formal method, it imposes some restrictive semantics. Parallelism is weakly accounted for. For example, the semantics of UML for the *join* node prevents interruptions during the evaluation of the value specification for the incoming flow. However, the semantics of this node does not consider the notion of the collision between inputs and outputs. Also, the means for input entrance (such as pins or ports) necessary for control nodes are not supported. That is, ports allow distinguishing between multiple inputs arriving through the same flow or different flows. Specifications for such concurrent dynamics are key for the *join* node and other nodes. Our work uses the formal DEVS modeling method with a sound simulation framework to examine concurrency of inputs and outputs relative to their handling in the external and output functions in DEVS-Suite.

The semantics of UML activities and actions, in particular, are also studied from



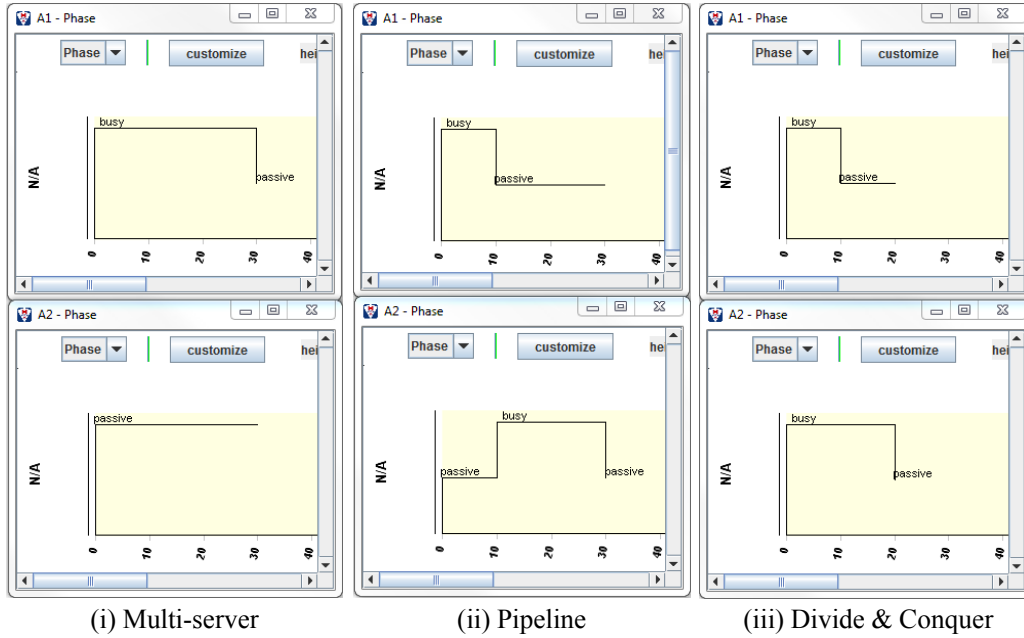
(a) An Animation View at the Initialization of Multi-Server Architecture (Figure 7.3a) In DEVS-Suite Simulator.



(b) I/O Trajectories for the Multi-Server Coupled Model (Left), and Time Trajectories for the Decision Atomic Model (Right), Note That Sending Phase Only Shows in the Superdense Time Trajectory Because It Is a Transitory State.

Figure 7.4: A View of the Architectures and Some of Their Corresponding Behaviors.

the theoretic vantage point (Crane and Dingel, 2008). A modeling language choice is Petri nets (e.g., Störrle and Hausmann, 2004). That is, activities can map to the Petri nets concept and elements. For example, Muram *et al.* (2014) defines a transformation of activities to Linear Temporal Logic (LTL) and then uses the semantics of Petri nets for execution. Such works do not account for arbitrary inputs nor behaviors. They target the verification of specific properties for the workflow (modeled as a flow of tokens as in Petri nets) or symbolic model checking. In other words, activities



(c) The Time Trajectories for the Phase in Each Atomic Model a1 and a2 in the Three Multi-Processor Architectures When Receiving One Job at the Beginning of the Simulation Only.

Figure 7.4: A View of the Architectures and Some of their Corresponding Behaviors.

can support rich behavioral expressiveness in terms of data and control flows with designated modeling elements and relationships. These constructs are not directly accounted for in Petri nets, LTL, or DEVS. Our work proposes using Parallel DEVS for enriching the parallelism semantics of activity models.

7.5 Conclusion

The DEVS formalism supports systematically developing complex, hierarchical dynamical models. It can be used to create activity models that can then be executed using a compliant Parallel DEVS simulator. We proposed introducing the concept of state to action. Thus, along with atomic model definitions for control nodes, we examined the use of the Parallel DEVS formalism for enriching the semantics of the

semi-formal activity modeling. We demonstrated key aspects of parallel processing of a collection of jobs in multi-processor architectures by showing their counterpart abstractions as activity nodes and then discussed their semantics. The DEVS-Suite grounded in system-theoretic modularity and hierarchy supports modeling and simulating reactive dynamics with arbitrary timing, which is used to study parallelism semantics of activity nodes. This is demonstrated through input, output, state behaviors (shown as superdense time trajectories) for the archetype multi-server, pipeline, and divide-and-conquer architecture that are simulated and examined. Further work includes a closer examination of activity model syntax and semantics given formal discrete event modeling methods and abstract simulation protocols.

Chapter 8

MODEL-DRIVEN TIME-ACCURATE DEVS-BASED APPROACHES FOR CPS DESIGN

In Cyber-Physical Systems (CPS), as in other complex systems (Simon, 1962), the system can be recursively built upon smaller parts that are much simpler to develop, operate, and maintain. This method of incremental construction ultimately is aimed at achieving correctness through restraining behavioral complexity (Sztipanovits *et al.*, 2011; Derler *et al.*, 2012). In the literature, concepts and formalisms are extensively discussed to establish the basis for systems to be designed in such disciplined and accurate manners utilizing modularity. Some formalisms allow models to be specified separately or collectively based on component and composition concepts (Zeigler *et al.*, 2000; Giese and Burmester, 2003; Alur, 2015). These definitions, as well as their other corresponding incarnations, are shown to be key for designing CPSs (for an example see Mosterman and Zander (2016)).

Inherent in any CPS is substantial interaction among decision points in some computational world and their interacting components in the physical world. The nature of such a relationship is tight, making the flow of information central in both directions. One direction is from computational to physical. The other one is from the other way around. As a result, coordinated interactions must satisfy both logical and physical rules. The immersion of computational consequences on the physical environment denotes one direction of the relationship while the other direction is indicated by the information observed in the computational parts. The relationship, in its broader sense, is considered from multiple perspectives besides the direction. Multiplicity and containment are two important examples of relationship properties

by which the complexity of such systems can be determined. In the context of CPS, a specific type of relationship is also considered where one end is the physical entity. The broader properties can be specialized in this particular type to ensure the correctness, especially for critical interactions.

Timing is a crucial aspect of CPS. Significant difficulties arise in CPS due to the physical nature of time (NIST, 2016), which is inherent and yet cannot be strictly or adequately controlled. Operations of some computational and physical entities are not inclined to the isolation of uncontrollable phenomena as physical time passes. The impact of the overall performance of system components can be affected to a more considerable degree relative to the variety of conditions that are accompanied by the timing specifications. Any breach of the timing agreement between the heterogeneous entities of the system may reveal threats to the entire system and therefore pose further difficulties.

The seclusion of timing in modern software as well as hardware systems has led to a significant deficiency for time-critical CPSs. Much of the process design is currently performed on the basis of as fast as possible execution with as much time granularity that can be afforded. Therefore, significantly abstracting out the time aspect is evident to achieve an optimal or even a satisfying result. However, in the critical stages of system design, including validation, some computation may turn out to be not useful and possibly at the expense of some others. A late execution may not be valid and may even cause serious damage. The validity of taking unsanctioned actions may not hold under some timing constraints.

Several concepts have taken place in the theory of modeling and simulation to inherently account for timing needs. These concepts, such as elapsed time, deadlines, and time intervals, are very beneficial. And their importance increases symmetrically relative to the cruciality of the CPS. In this work, we attempt to work on the assimila-

tion of the system-theory definitions toward better accommodation of modern system design concepts as manifested by the CPS. An action-level modeling approach is introduced with an emphasis on actions constrained with time-invariant for real-time environments. The conformance of the developed models is improved through meta-modeling in which the higher concepts are addressed at a higher level of abstraction when possible. A model-driven approach is then attained for the behavioral specification for model components at the individual and composite levels. The proposed approach makes a clear distinction between actions for the logic associated with the computational entities and their interactions with physical entities.

Thus, it is possible to characterize actions in the CPS to appropriately account for their consequences. Overall, actions may perform at any point in time for different purposes subject to adequate and possible time granularity. Some of these actions only take place in the computational part of the system, others in the physical region. An example for the former can involve any pure computations. The latter can affect physical operations and responses. Another set of actions includes the interaction between the computing environment and the physical parts. We characterize these actions to be actuating and sensing actions. This set of actions is crucial in the context of CPS since they deal with coupling between computational and physical parts, as shown in Figure 8.1.

We begin by presenting some of the necessary backgrounds about the underlying formalisms of this work. That is, a brief background is given about P-DEVS, RT-DEVS, and ALRT-DEVS, all of which are targeted for simulation. To support verification, instead of validation, the Finite-Deterministic DEVS (Hwang and Zeigler, 2009) is developed. FD-DEVS is a DEVS variant for model-checking. More recently, Constrained DEVS is developed. It targets the underpinning non-deterministic and stochastic aspects of CPS (Gholami and Sarjoughian, 2017). The background is also

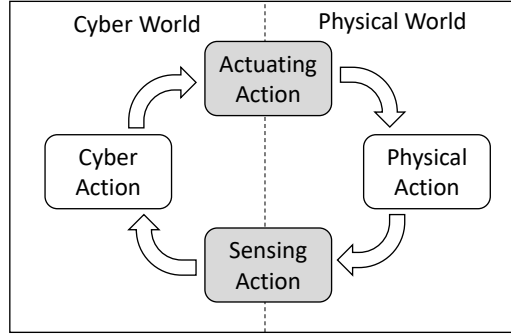


Figure 8.1: Actions in the CPS Are Characterized into Four Types. the Types in Grey Are Crucial from a CPS Standpoint since They Are Akin to the Tight Coupling between Cyber and Physical Parts. Actuating Actions, for Example, Can Impact the Physical Environment Directly and Therefore Their Consequences Are Critical.

given for timed automata formalism. We follow that with a discussion of the present works with a focus on model-driven DEVS-based methodologies for addressing similar problems that can apply to CPS. Then, we present the action-level specification for modeling CPS. Before concluding, we demonstrate the approach by modeling the dynamics of a traffic intersection with multiple relays and a controller with a discussion on the verification.

8.1 Background

There are many extensions of classic DEVS. In every one, the extensions and variants maintain some of the key concepts and properties while extending or replacing them for specific needs. A prime example is parallel-DEVS, where atomic and coupled models can execute simultaneously as compared with classic DEVS. Several extensions have taken place for the time advance function. The aim is to provide capabilities such as real-time Real-Time DEVS (RT-DEVS) which uses time-window (aka Time Interval (TI)) (Wang and Cellier, 1990) and actions. This work relies on Action-Level Real-Time DEVS, which introduces real-time statecharts for modeling actions.

We will describe these briefly as well as other related formalisms in the following sub-sections.

8.1.1 *Parallel DEVS*

The set-theoretic specification of the atomic model is an abstract representation of a system component. The formal specification can be defined independently of any specific platform, language, and simulator. The Parallel DEVS (P-DEVS) was proposed by (Chow, 1996) to provide both conceptual and execution benefits for the modelers. The basic formalism of P-DEVS model is an algebraic structure – atomic model = $\langle X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$. X is the set of input events. S is the tuple of sequential states with at least two variables which are *sigma* (σ) and *phase*. Y is the set of output events. δ_{int} and δ_{ext} are the internal and external transition functions, respectively. δ_{con} is the confluent transition function, which can be specified to handle the collision between external and internal events. λ is the output function which transforms S into Y at specific time instances. ta is the time advance function which maps the internal state into a positive real number using elapsed time since last state transition.

8.1.2 *Real-Time DEVS (RT-DEVS)*

8.1.3 *Action-Level Real-Time DEVS (ALRT-DEVS)*

(Sarjoughian and Gholami, 2015) proposed this extension to support defining real-time constraints at the action level from both modeling and simulation point of view and assisted with the concept of locations identified for real-time statecharts. That is, the modeler will be able to develop a real-time model. Under certain conformance conditions, the real-time simulator will simulate this model. The simulator extends

the abstract simulator for the real-time software system. It fundamentally lies based on the parallel and real-time DEVS as well as real-time statecharts.

The notion of time in ALRT-DEVS is defined based on both theoretical and pragmatic perspectives. A unified concept with formalized specification for logical-time, real-time and physical-time underlie ALRT-DEVS formalism with algebraic structure – atomic model = $\langle X^b, Y^b, S, A, \Gamma, \Omega, \psi, \lambda, ti \rangle$. The time in models, as well as their simulators, is concertized according to a physical clock where the distinction between physical-time, real-time, and logical-time, is well-established. The physical time denotes the time in the actual (physical) environment in which infinite accuracy and precision exist. All other timing schemes include a physical signal (NIST, 2016). Therefore, real-time is an approximation of physical-time but not equal to it due to uncontrollable factors in computing platforms. The logical-time is an abstract computable quantity to provide the basis by which the software logical clock can proceed increasingly. It only ideally has the properties of the physical clock.

First, state variables are characterized to be primary and secondary variables. The primary state variables are the *phase* and the sigma σ by which the next state is determined in P-DEVS models generally. The secondary state variables are defined as needed to denote for specific system dynamics. The notion of location is therefore defined to enable a different kind of transition on the basis to the state change, whether it is associated with a primary or secondary state variable. Transitioning between different locations is usually associated with which guards are specified in terms of the secondary state variable. Actions, which are the fundamental units in the scope of the current work, can be individually specified for locations.

8.1.4 Timed Automata

The theory of timed automata (Alur, 1999) explicitly admits the notion of time by which it becomes suitable for the modeling and analysis of real-time systems as opposed to basic logical model checking. The behavior of such systems can be modeled in state-transition formal notations. The notation is then annotated with timing constraints using clock variables. Further restrictions can be therefore imposed on the state space to allow for the verification of some system properties under the given timing constraints. A transition system is defined by a set of states, a set of initial states, a set of labels or events, and a set of transitions. The timing constraints are then introduced with a finite set of real-valued clocks for a finite graph where the vertices are called locations, and the edges are called switches. The locations are associated with some time-invariant to constrain the elapsing of time in that location. The switches are instantaneous.

8.2 Related Work

There have been several DEVS-based approaches that are suitable for use in CPS. Sarjoughian *et al.* (2013) proposed a new model for interacting an ALRT-DEVS with a physical system. The simulation is composed of a computational-physical system. This kind of system can be considered a cyber-physical system if it offers the designated capabilities for CPS. The DEVS-Suite 3.0.0 is extended to support the capacity of communication between computational and physical parts of the system. An experiment is devised with a tight coupling connection to ensure the validity of the model under hard real-time constraints. The connection between a 4-relay Phidget and real-time simulation is thoroughly conducted under different settings to examine the turnaround time for the switching actions. The role of reviewing

such a hard constraint is complementary to the logical-time restrictions. Therefore, together, they form a stronger basis for carrying out different experiments about the CPS under study.

Many other works do not directly address CPS. However, they can be utilized in that direction since the problems, in which they try to address, are akin to their counterparts within the CPS context. In Risco-Martín *et al.* (2016), although the work is not directly targeting CPS, the authors propose a model-driven hardware-in-the-loop method to obtain embedded hardware starting with their software representations incrementally. The methodology depends on the DEVS formalism. Instead of extending the simulator, as proposed by Hong *et al.* (1997), the hard real-time constraints take place in the parent formalism through star models. Star models are an interface for atomic models to allow for building abstract models for the concrete hardware ones based on the concept of the transparent simulation environment. The elevator circuit real-time model (Zeigler *et al.*, 2000) is designed and implemented with the adder as a HIL component.

Nonetheless, the major problem persists in modeling CPS notwithstanding the ongoing efforts in languages, notations, and tools (Derler *et al.*, 2012). In digital hardware, time-accurate modeling is necessary to examine digital designs at a fine-grained resolution. The CPS sensitivity to such requirement poses challenges. Time-accurate approaches are significant in the modeling of CPS. Regardless of the usefulness of modeling languages such as (OMG, 2012, 2018), the missing semantics and the weaker notion of time are two primary causes for not using them in time-critical system design.

8.3 Action-Level DEVS Specification Using Activity Modeling

Previously, we proposed establishing a DEVS foundation for activity modeling and simulation (Alshareef and Sarjoughian, 2017). However, the time notion was limited to support the simulation step to somehow correspond to the debugging step. We extend this notion of time to be supported at the action level. That is, the action can be defined with time constraints to enrich the activity modeling further toward time-accuracy.

8.3.1 CPS Activities Metamodel

The metamodel of the UML activities (OMG, 2012) is circumscribed and then extended with necessary definitions to elevate the support for the concept of the state. The path for doing such can be through providing the basis for their conformance to the DEVS formalism at a higher level. The metamodel consists of three major elements. The first element is the action in its broader sense to support modeling at the action level. The second major element is the control node to support defining control logic. The last but not least is the activity edges where they can also be specialized to be control and object flows. Their mapping to DEVS has been discussed thoroughly in the previous work (Alshareef and Sarjoughian, 2017).

We note that the performed process is not merely transformation from one form to another, but rather, grounding activity modeling with the rigorous formal specification. We argue that the DEVS formalism is a suitable candidate for this purpose. On the one hand, the definition of state with a strong notion of time can take place as a fundamental basis for the proposed modeling approach. On the other hand, the modeler can also benefit from the behavioral modeling constructs that are provided within the activity metamodel. Therefore, the action exhibits as an abstraction of its

corresponding atomic model (Alshareef and Sarjoughian, 2018b).

The action specializes from the activity node, which also defines the super-type for the control nodes. The control and object flows are both specialized from the activity edge. The edges are instantaneous. For the action, we define time boundaries based on DEVS temporal structure. The ongoing action can have an elapsed time. Therefore, it can be interrupted at any point in time upon receiving some external input events. The elapsed time also indicates the completion of the action. These boundaries are defined in terms of the time advance function. Their corresponding values at the time base may differ without violating their time-invariant. Such difference is crucial, especially in the modeling of CPS since actions may not always complete, and therefore, further considerations should take place during the modeling process. With the existing DEVS metamodel (i.e., Sarjoughian and Markid (2012)), we can incorporate these definitions with the DEVS metamodel at the higher level (see Figure 8.2).

8.3.2 *The Modeling and Simulation of a Traffic Intersection*

The process of interest is a traffic intersection where multiple cars may approach the intersection from various directions. Many crucial time-sensitive requirements have to be accounted for in this model in different modeling environments. From a physical point of view, cars can approach the intersection from one and only one direction. There are also temporal logic requirements as well where the car has to approach the intersection before it enters it. Such hard real-time constraints would go beyond the logical constraints toward the physical environments and the encountered limitations when interacting with physical aspects. Such issues arise from phenomena like time latency from dispatching events to a physical relay until getting the acknowledgment back. Each of these aspects is crucial at the design stage of the

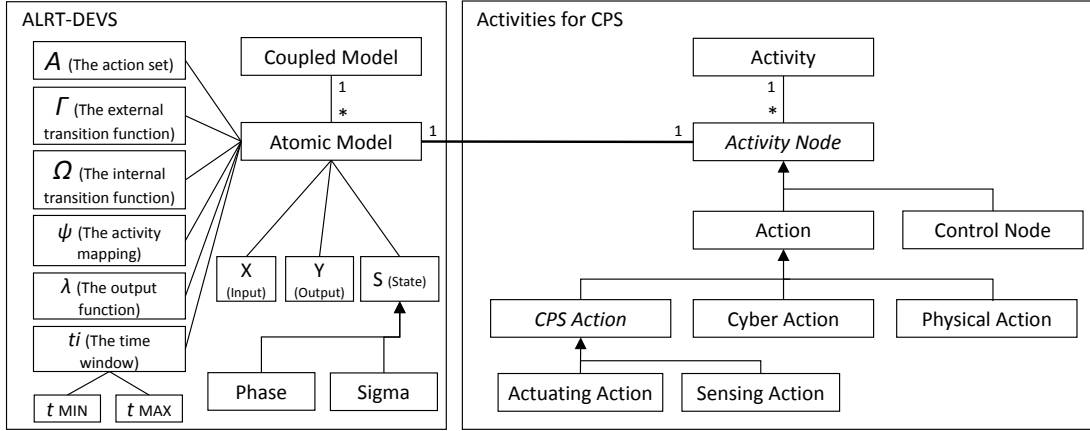


Figure 8.2: The Activities Metamodel Is Circumscribed and Extended with CPS Action. ALRT-DEVS Metamodel Is Also Linked with the Activities Metamodel at a High Level to Establish the Grounding for the DEVS Modeling and Simulation of the CPS Activity. The X, Y, and S Sets Are the Same as Those Defined for P-DEVS. Some Cardinalities Are Visually Omitted. The Elements with Italic Are Abstract Super-Type Elements.

CPS.

We have covered the logical aspects at the activity level in the previous work. We now discuss the temporal ones to some extent. Then we present a real-time extension of the approach for the subsequent Section 8.4.

In Figure 8.3, we simplify the process of the traffic intersection by creating yet another abstraction of it at the action-level in the collective activity. The actions are timed in such a way to ensure safety by accounting for ramifications. The actions of approaching an intersection can be in some active state in parallel. However, we assume that the crossing must allow the flow for one direction only. Otherwise, an accident happens and gets reported to the monitoring model after that. Parallel entrance to the intersection can happen only for vehicles approaching from the same direction. These constraints are mere examples, and yet further elaborations can

be made by the modelers throughout the model development life cycle. The goal is to establish the basis to take such models and interpreting them by the simulator to conduct the necessary analysis and verification based on their specifications. The state-space of the coupled model consists of all permutations for all the possible states of the actions thereof. We will discuss some possibilities of verifying such models in Section 8.5. For the simulation, after the activity, it gets interpreted as an activity digraph in the DEVS-Suite simulator (ACIMS, 2017b) (see Listing8.1). This code snippet shows reading activity nodes and instantiating the atomic models after that.

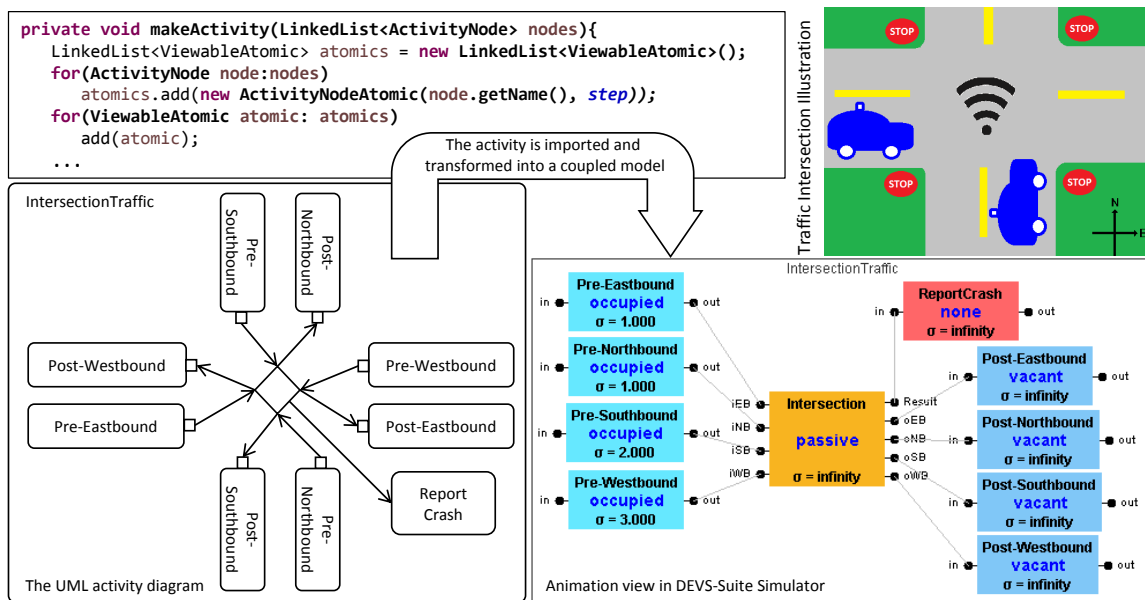


Figure 8.3: The Activity for Modeling Traffic Intersection and Simulating It In DEVS-Suite.

Listing 8.1: The activity interpretation in DEVS-Suite elaborated with decision and action nodes.

```
for(ActivityNode node:nodes){
    ViewableAtomic atomic;
    if(node.getType().equals(NodeType.DECISION))
        atomic = new DecisionAtomic(node.getName(), step);
    else
        atomic = new ActionAtomic(node.getName(), step);
    atomics.add(atomic);
    map.put(node, atomic);
}
```

The semantics of the decision node has been specified in its general form for the UML. Therefore, it is a domain-specific abstraction in the P-DEVS, which yet generally represents the corresponding semantics of the decision node in handling incoming flows. First, the atomic model initializes in a passive state for an unbounded time. Then, upon receiving input events, the model reacts to these input based on the condition associated with the incoming flow which maps into a coupling and input port in the corresponding atomic model. After checking the input and the condition, possibly along with the other state variable, the next state is determined. The output is sent out after that when applicable, and then the internal transition is performed.

This kind of modeling and simulation falls into the realm of methods for analyzing and designing CPS. We consider the modeling and simulation by Damodaran and Mittal (2017) and Alur (1999) to be relatively part of this taxonomy. These approaches focus on logical aspects and temporal logical ones, whether it is based on the DEVS formalism, such as Damodaran and Mittal (2017), or on the notion of clock variables with real values, such as Alur (1999).

8.4 Interacting with Reactive Computational-Physical Systems

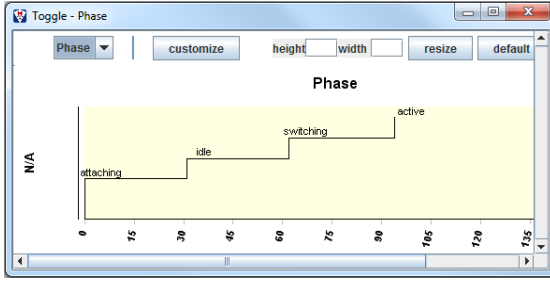
One of the significant challenges in modeling CPS is the integration of the notion of time with the current frameworks and tools (Derler *et al.*, 2012). ALRT-DEVS (Sarjoughian and Gholami, 2015) uses the idea of time windows to recognize the uncontrollable factors in which an imperfect physical environment operates. In the following work (Sarjoughian *et al.*, 2013), the focus is on the actuating actions within the context of a CPS action. The experiment was conducted to carefully investigate the turnaround time of the performed action on the physical entity, which is a 4-relay Phidget. The experiment has been conducted within multiple settings to examine the property in real-time. We note that this taxonomy of modeling is distinct from the previous one since it focuses on dissecting the real-time properties during the run-time of the simulation with real-time constraints. That is, TIs are introduced at the action-level of the modeling. They become enforced after that by the extended simulator where the synchronization between the cyber and physical actions takes place. In a sense, it is cyber-physical modeling and simulation for CPS, which can be suitable for simulation with hardware-in-the-loop. It does not restrict to the computational aspect of CPS. The time constraints are strictly enforced on the actuating and sensing actions. For example, an actuating action has to take place within a certain time window. Otherwise, it is considered to be invalid. Missing the execution of actuating actions, as well as sensing ones, may result in losing the fidelity of the model depending on the critical situation of the system. Some systems may have less or more tolerance than the others based on the domain and potentially other variables in which the time can play a critical role. The important aspect from a modeling perspective is that such properties are rigorously accounted for in a formal specification and well-formed models.

Since we look closely into actions, this taxonomy applies to the CPS actions, that is, the actuating and sensing actions where interactions between the computational and physical worlds take place. The time windows enforced on these actions given some time restrictions (i.e., time invariants are specified for these actions). We note that a connection can take place with the taxonomy discussed in the previous section (see 8.3.2) to enable the actuating and sensing actions to take place in real-time. For example, the actuating actions can be employed for the intersection to control the relay of the traffic flow or signals by sending actuating actions. Also, the sensing actions can take place by detecting the moving vehicles toward the intersection to perform the necessary computations and send the corresponding actuating actions after that. Both types of actions can only take place under hard real-time constraints. Figure 8.4 shows different phase trajectories of toggling the relay as a CPS action. The time granularity is in milliseconds. The TI is defined with $t_{min} = 0$ and $t_{max} = 40$. Thus, the transition to active succeeds in Figure 8.4a because the TI is met when the switching takes place after 32 milliseconds. In Figure 8.4b, the TI is not met; therefore, the switching fails. These scenarios are produced using 4-relay Phidget InterfaceKit-0/0/4.

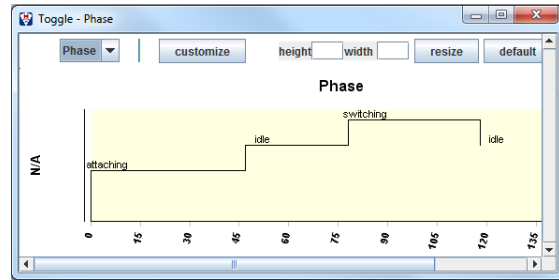
8.5 Verification of the CPS Activities Models

The verification of CPS activities manifests to some degree by significantly bounding the state space that defines a model's dynamics. A key goal is to sacrifice, as little as possible, both rigor and expressiveness of the model as measured. Although this places restrictions on the degree to which a model can be simulated, this leads to achieving formal verification through model checking (Gholami and Sarjoughian, 2017).

In such a simulation-based verification approach, the constructed model can help



(a) A Scenario Where the Time Window Is Met When Switching Therefore Transition to Active.



(b) Another Scenario Where the Time Window Is Not Met When Switching.

Figure 8.4: Phase Trajectories for Different Scenarios for *Toggle* as a CPS Action.

ensure various temporal properties, particularly considering that cyber-physical systems are non-deterministic and stochastic. The goal is to use modeling of activities given a set of a priori defined atomic models behaving as expected. It also enables the use of other verification techniques. Many scenarios are drawn to verify the behavior of the activity after being interpreted by the DEVS simulator. For example, in the traffic model (see Section 8.3.2), we can check if the model corresponding to the decision node reports a crash when multiple cars approach it in the same direction. This scenario indicates that the model is not behaving correctly according to the problem specification. We also can check if the car gets directed to the correct destination after passing through the intersection. We can also check if the crash gets reported by directly injecting two vehicles into the intersection at the same time. These scenarios are checked, for example, by thoroughly disciplined experimentation. The DEVS-Suite simulator uses and expands the notion of experimental frame (Rozenblit, 1991) where defined temporal properties can be verified.

8.5.1 Reasoning About Temporal Behavior

We aim to build an integrative environment to allow the development of activities as standalone behavioral models and also enable further reasoning capabilities about them. Reasoning can take different forms. However, it is quite common to be used in the form of model checking to facilitate various forms of knowledge representation. From this point of view, knowledge is represented in the form of a DEVS model. On the other hand, the reasoning capabilities are also used to perform the model checking after imposing restrictions on the state, timing, ports, and external/internal events.

Thus, the history of the causal effects is computed using an Answer Set Programming (ASP) tool (Bartholomew and Lee, 2014) for a fixed integer m that represents the length of the history. The tool itself aims at the first-order logic; however, the introduction of history makes it possible to deduce about some temporal properties. The formulation of the ASP program takes into consideration bounded ranges and sends them out to the model as well as the transducer. An example of that could be the *phase* state variable of the intersection and some direction toward it. The rule that describes the causal effects considering the temporal aspect can be formulated as $i : phase(pre - eastbound) = occupied \rightarrow i + t : phase(Intersection) = occupied$ where i is the timestamp, and t is determined time advance for the phase *occupied* in the pre-eastbound atomic model. This rule can be examined within some defined finite m under the imposed restrictions on the state space and specifically the time advance. Other properties can also be checked similarly. Outputs of the ASP program are generated and fed to the model. Then the transducer checks for the verification. The following rules describe the effects approaching from the pre-eastbound (PE) and pre-northbound (PN) actions and their rules for the intersection node (I):

$$i + t : phase(I) = occupied \leftarrow i : phase(PE) = occupied$$

$$i + t : \text{phase}(I) = \text{occupied} \leftarrow i : \text{phase}(PN) = \text{occupied}$$
$$i + t : \text{location}(x) = I \leftarrow i : \text{location}(x) = PE$$
$$i + t : \text{location}(x) = I \leftarrow i : \text{location}(x) = PN$$
$$i + t : \text{phase}(Crash) = \text{active} \leftarrow i : \text{location}(x_1) = I \wedge \text{location}(x_2) = I \wedge x_1 \neq x_2.$$

Based on the previous rules, when t is one, we can formulate questions about some basic facts such as:

$$\left[1 : \text{phase}(Crash) = \text{active} \wedge \left(0 : \bigvee_l \text{location}(x) = n \right) \right] \rightarrow 0 : \text{location}(x) = I.$$

The rules and the entailment question can be both represented in the language of F2LP (Lee and Palla, 2009). Future work is considered on providing some action-level verification capabilities for behavioral models that are specified while recognizing the notion of action.

8.6 Conclusion

Substantial effort is required to bring definitions from the DEVS formalism and its variants along with their underlying simulators and model-checkers for analyzing and designing cyber-physical systems. We expect that the process can benefit from employing model-driven approaches where they have been useful in multiple occasions for DEVS modeling within different variants (Cetinkaya *et al.*, 2011; Moallemi and Wainer, 2010). The promising capabilities of the employment of Model-Driven Engineering (MDE) concepts can be achieved with proper conformance to the Model Driven Architecture (MDA) meta-layers and the DEVS formalism. A combination of MDA and DEVS stands to benefit the process of model development by accounting for some domain-specific knowledge added to the general-purpose DEVS model abstraction (Sarjoughian *et al.*, 2015). In this work, our attempt can be viewed as a contribution to the effort required to concretize behavioral abstractions. Therefore, further effort is required to account for activity-based behavioral specifications

relative to the MDA modeling layers.

We have demonstrated how the design of CPS can be approached with the help of activity-based modeling incorporated into system-theory and DEVS in particular. The use of the P-DEVS formalism and its underlying simulators as a platform for CPS is effective due to the inherent timing and modularity enabled by benefiting UML behavioral activity modeling and grounded with ALRT-DEVS modeling formalism. In contrast, many existing approaches do not account for the notion of time intrinsically and therefore leading to possible inconsistency and conflicts unless these issues are resolved at the implementation level. Furthermore, modularity can also serve as a means for the scalability at the structural and behavioral specification of cyber-physical systems. Such systems have strong non-determinism and stochastic traits.

Another significant advantage of this work is overcoming the issue of ending with a large fUML model that is hard to develop. This problem arose in the (OMG, 2018). It is the reason for not creating the models of the execution model in activities. Instead, they have been specified in their corresponding Java code because significant activities quickly become too large to handle. Recent work has been proposed to address it (Bedini *et al.*, 2017). Richer fUML models tend to grow large, which may lead to other issues concerning scale. As discussed in the traffic example, the activity models have been significantly richer to handle complex time-critical dynamics in the traffic model with relatively fewer elements. This improvement is due to the DEVS formal grounding.

METAMODELING ACTIVITIES FOR HIERARCHICAL COMPONENT-BASED MODELS

Models can be perceived in different ways, especially at a meta-layer. It is useful to examine various models and metamodels as representations of some components. However, issues may arise due to using different representations and possibly at different layers. Model-Driven Architecture (MDA) has been proposed (Soley and the OMG Staff Strategy Group, 2000) to provide a framework to work with abstract and concrete models across a four-layer architecture. It defines a semi-formal approach to guide the process of developing models and metamodels across these layers and the relationships between different layers and within the same layer. Essential relationships are instantiation, interpretation, conformance, and transformation.

Metamodeling is used to provide a means to describe systems in general (Henderson-Sellers, 2012) and the relationship between systems of systems. A key to metamodeling is to identify and relate essential artifacts of a system of interest from higher-level abstractions. The problem is evident because these abstractions are representatives of their corresponding realizations to some lower-level abstraction and eventually in the implementation space. Multiple understandings arise when metamodeling certain elements of the system. As such, handling differences becomes quite challenging, especially when system complexity is high or partially known. Representing instances raises issues in metamodeling too. According to (OMG, 2018), the notion of instantiation is only meaningful within a metamodel. The relationship across meta-layers is defined through the interpretations of the model from the higher layer in the immediate lower layer.

The model itself is one of the four basic entities in the modeling and simulation (M&S) framework (Zeigler *et al.*, 2018b). It forms the fundamental component with a sound mathematical foundation. The model is conceived as a set of instructions by a simulator. Therefore, it simulates the model and correctly produces its state and output trajectories based on the input trajectories for a given initial state. According to the I/O requirement for deductive and deterministic models (Wymore, 1993), one unique output trajectory is generated for each input trajectory and initial state. As such, the definition of the model has a precise semantics, which is not the case in the existing semi-formal methods such as the MDA and the Unified Modeling Language (UML) (OMG, 2012).

Actions and states are two commonly used concepts when ascribing system behavior. They provide suitable means for abstractions about behavioral system specifications. However, their semantics may pose challenges when they are considered at a high layer in the MDA hierarchy. A key aspect of the problem resides in having a way of interpreting states and actions into concrete system models. The relationship between action and state can collectively formulate a basis for how their definitions and specifications are used together to describe system dynamics. We can examine such relation from the individual as well as integrative standpoints. In both ways, the complementary perspective of their roles is taken to account for richer behavioral specifications. Neither one is known to be sufficient to produce proper manifestations at the implementation level.

This work focuses on the trio of component, state, and action, in formulating abstractions about complex systems. We examine relations between action and state at a higher level, from one side, and their possible corresponding representation in the theory of modeling and simulation, on the other side. We also discuss the relation between the two abstractions themselves, the action, and the state. The discussion is

grounded in the context of metamodeling activities where the role of action collectively serve with control toward behavioral specifications of the component-based models.

In the following section (Section 9.1), we discuss the notion of component modeling in the context of this work. In Section 9.2, we describe the server systems as perceived in the system-theoretic literature. In Section 9.3, we discuss behavioral specifications across the different layers for metamodeling activities. We discuss the related work and the concluding remarks in the Sections 9.5 and 9.6, respectively.

9.1 Component-based Modeling

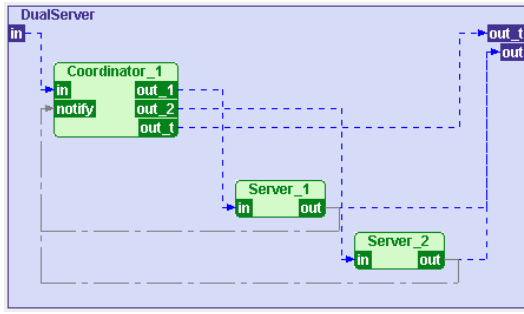
As much as we are concerned about identifying components concerning their structure, it is also essential to understand their behavior by which system dynamics are defined. Components at the higher level can bear hierarchical relationships with other components. An example is the template model (Zeigler and Sarjoughian, 2017). Some instance template models can extend the template model. Instance template models can transform into instance models after that. The relationships between the model mentioned above types allow modelers to incrementally develop models while obtaining simulations after adequately providing their specifications.

Specialized components define a type of relationship where some components share and mimic the structures and behaviors of some other components. Therefore, a component specializes in another, such that, it redefines its structure and behavior. As properties, functions, and relationships add up to a component, the component becomes a more accurate representation of its counterpart in the real system. After that, the instance model type transforms the instance template model and choose all specialization relationships. The resulting model can have multiple components via decomposition.

In Figure 9.1, a hierarchical component view is designed using the Component-

based System Modeling and Simulation (CoSMoS) (ACIMS, 2017a). The tool’s framework provides a means by which components can be defined along with their couplings, whether they are internal, external input, or external output coupling. We can define both primitive and composite models as components with possibly composition and specialization relationships. Considering the Discrete Event System Specification (DEVS) formalism, primitive and composite models correspond to atomic and coupled models, respectively. Atomic model functions are the state transition functions including the internal (δ_{int}) and the external (δ_{ext}) transition function, the output function (λ), and the time advance function (ta). The confluence function (δ_{con}) is defined to exploit parallelism in the Parallel DEVS formalism further. In CoSMoS, such behavior can be defined using a variant of statecharts devised to incur certain aspects of the behavioral specifications in a state-based manner (Fard and Sarjoughian, 2015). The coupled model is a composite component that has a finite number of components thereof along with input and output ports and couplings. An external input coupling defines the coupling between an input port of a coupled model and an input port of one of its components. An external output coupling defines the coupling between an output port of a contained component and an output port of a coupled component. An internal coupling resides between two components of a coupled model. The DEVS formalism is modular such that communication between components is only allowed through coupling and their designated ports. The component receives input or bag of inputs arbitrarily, and after that, it may only accept them according to the correspondent specifications.

The dual server composite template model is devised to include three components, one instance of the coordinator template model, and two instances of the server template model. We define couplings as shown in Figure 9.1a. The components are identified to mimic the dual server system, such that, the job is assigned to the first



(a) The Components, Ports and Couplings for the Dual Server Model.

| Attribute | Value |
|-----------------|------------|
| Model Name | DualServer |
| Model Type | COMPOSITE |
| Children | |
| Immediate | 3 |
| Total | 3 |
| Ports | |
| Input | 1 |
| Output | 2 |
| Total | 3 |
| Couplings | |
| Internal | 4 |
| External Input | 1 |
| External Output | 3 |
| Total | 8 |

(b) The Structural Metrics for the Dual Server Model.

Figure 9.1: Modeling the Dual Server System in CoSMoS.

server if it is available. Otherwise, it gets assigned to the second server. If both servers are busy, then the job is added to a queue to wait for one of the servers to become available.

We define the dual server coupled model with one input port for receiving jobs. We also define it with two output ports. One port is sending notifications about the job dispatched to its corresponding server, and another is for producing the completed jobs. The coordinator template has two input ports and three output ports. An input port is for receiving jobs, and another port designated for receiving a notification when the job gets completed which means that the corresponding server becomes available. Besides, the coordinator has three output ports. Two of them are for sending output to servers. The third one is coupled with the dual server out_t port, which is for experimental purposes. The server template is defined with one input port for receiving assigned jobs and one output port for producing them after finishing their service. An internal coupling is defined for each server output to the *notify* input port of the coordinator. A notification goes through this coupling to coordinator to highlight job completion. It consequently indicates the corresponding

server availability.

The formal specification of the dual server coupled model is the following:

$$\begin{aligned}
A &= \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC \rangle, \text{ where} \\
InPorts &= \{in\}, \\
\text{where } X_{in} &= V \text{ (an arbitrary set), } X_M = \{(v) | v \in V\}; \\
OutPorts &= \{out, out_t\}, \\
\text{where } X_p &= V, Y_M = \{(p, v) | v \in V \text{ and } p \in OutPorts\}; \\
D &= \{coordinator_1, Server_1, Server_2\}; \\
M_{coordinator_1} &= Coordinator; M_{Server_1, Server_2} = Server; \\
EIC &= \{((DualServer, in), (coordinator_1, in))\}; \\
EOC &= \{((coordinator_1, out_t), (DualServer, out_t)), \\
&\quad ((Server_1, out), (DualServer, out)), \\
&\quad ((Server_2, out), (DualServer, out))\}; \\
IC &= \{((coordinator_1, out_1), (Server_1, in)), \\
&\quad ((coordinator_1, out_2), (Server_2, in)), \\
&\quad ((Server_1, out), (coordinator_1, notify)), \\
&\quad ((Server_2, out), (coordinator_1, notify))\}.
\end{aligned}$$

Some complexity aspects of this model are determined by looking into the components and the communication between them. Figure 9.1b shows the structural metrics after defining the model in CoSMoS. The metrics of components, ports, and couplings are easy to compute since all elements of the dual server persist in a relational database. Different coordination mechanisms could result in different measures. Moreover, different designs of the same coordination mechanism could also result in different measures. The ability to easily observe such metrics becomes important,

especially when such models become larger and ordinarily more complex. Models of such nature are developed to grow, especially when accounting for such complexity at the early stage of design. Further examination of such metrics is necessary to understand their meaning, if they have one, whether in domain-specific settings or in general.

The ability to be characterized with scale and complexity traits (Sarjoughian, 2017) is essential even though measures for behavioral characteristics may be taken qualitatively. Quantitative measures are usually taken for structural aspects such as the ones in Figure 9.1b. In the context of metamodeling activities, some metrics about syntactical properties of actions and flows could be easily obtained. More information can also be obtained about the input/output pins and control nodes. However, this information needs to be further examined to determine their relevance in contributing to the overall behavioral scale and complexity traits.

9.2 Coordinating Between Server Components

The server system (Wymore, 1993) consists of servers along with a queuing component that coordinates dispatching of jobs to one or more server components. The server system can accept inputs via three ports. The first port is for receiving elements of the queue. The second one is to accept the service time produced by the generator. For example, service time determines job processing according to some probability distribution such as the exponential distribution. The third port is added to receive a random Boolean to indicate the existence of an error to determine the result of conducting the processing operation. In the model shown in Figure 9.1a, we omit this port and, instead, we designate a port for the receipt of the job completion by a server. The job itself and the service time are both accepted via the input port *in*.

State variables for Parallel DEVS atomic models are categorized into primary and secondary variable types. The primary state variables are generally defined by *phase* and *sigma* (σ). In the coordinator component, they are used to determine how the model responds to the arrival of a job whether by storing it in a queue or sending it out to the corresponding server. In the case where some server is available, the *sigma* is assigned to zero *time advance* and therefore sends out the job to the selected available server. Otherwise, if there is no available server, the job is stored in a FIFO queue.

Also, the state of the coordinating unit includes several secondary variables. The received job is maintained along with its accepted service time in advance to assigning it to some designated server. The coordinator consists of a queue to hold on the received jobs in the case that all servers are busy. The stored jobs in the queue are also time-stamped by their arrival time to facilitate analyzing their turnaround time at a later stage. Another state variable of the coordinator is an array to maintain the availability status of each server. When a server is assigned a job, its availability status is changed to be false until receiving the job completion notification from it. It is changed to true afterward. The determined server *id* for assigning the job is also maintained to send out the job through its designated port and coupling. Hence, the number of servers is static and therefore known by the coordinator during the initialization stage. The current model does not account for structural changes during the simulation.

An activity model corresponding to the external transition function of the coordinator is depicted in Figure 9.2. This model is devised using our previous activity-based DEVS model specification approach (Alshareef *et al.*, 2016). It describes coordination and assignment of jobs to servers (see 9.1a). Inputs are received and then processed in an iterative manner by the expansion region. For each job arriving through the *in*

port, the coordinator model either assigns it to an available server or stores it in its queue. In the case of assigning a job to a server, the sigma set to zero and the phase set to *sending* for immediate dispatching of the jobs. For the inputs arriving through the *notify* port, the availability of each server is updated since each received input indicates processing of a job has been completed and therefore its assigned server becomes available. When a server is available and there are jobs waiting to be processed, then as many jobs as possible are removed from the queue and dispatched to available servers and marking them as unavailable. This type of activity is characterized with fine-grain nodes with details about the inner specification of the atomic model. The activities in the Figures 9.3 and 9.5 are collectively specified and therefore reside at different abstraction layers.

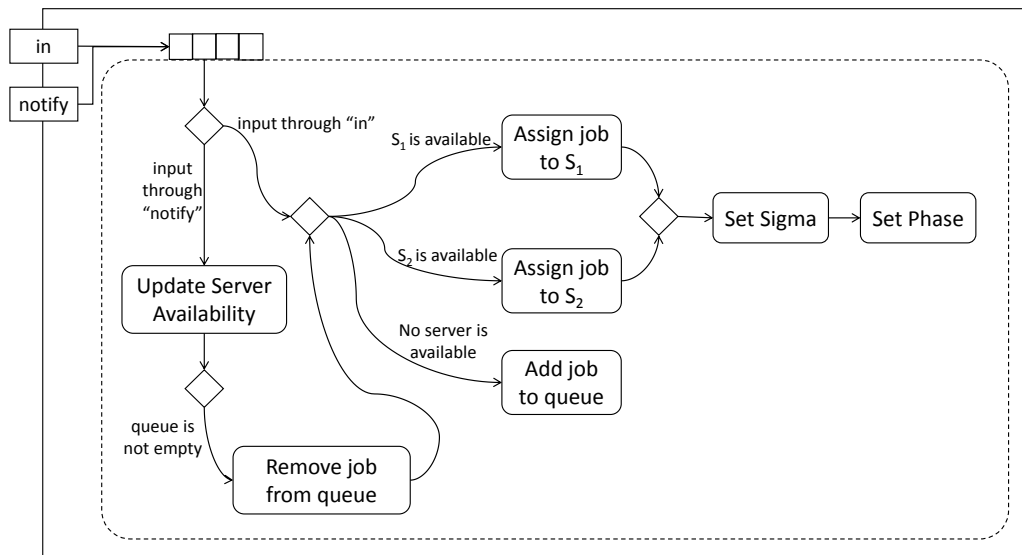


Figure 9.2: An Activity for the External Transition Function of the Coordinator.

The coordinating unit can consist of a component or multiple components of the multi-processing architecture system. Different coordination can also be considered to account for different processing architectures such as the pipeline and divide & conquer (Alshareef and Sarjoughian, 2018b). Changes in coordination can take place

through state definitions or manipulation of the primary or secondary state variables. They can also take place via different designs of the components and their couplings. The coordinator is a subsystem within a larger subsystem or the system itself. Such a subsystem can consist of one component or multiple components where they are all closed under coupling to collectively achieve the coordinating task. In the context of activities, parts of the coordinating task are delegated to a set of control constructs to arrive at the intermediary stage of modeling a different kind of processing architecture. For example, *decision* and *split* nodes can be used to enrich models at a higher level for logic descriptions of the coordination task for multiple processors. On the one hand, conditions of the outgoing flows from the *decision* node are employed to check for the availability of servers. On the other hand, dispatching outputs synchronizes for different servers such as in divide & conquer. They dispatch to the server and the transducer, such as in the multi-server architecture. This synchronizing, of such a task, takes place through the *split* node.

Inputs to such a system are generated externally and fed into it through the external input coupling. The flow synchronizes mainly at the arrival of inputs to the input port of the coordinator. It also synchronizes at the end of completing the job after service. Therefore, servers can operate independently, yet their outputs have to be synchronized. Such synchronization is depicted in activities by the *merge* and *join* nodes (the activity in Figure 9.3 is to be discussed in Section 9.3). Due to modularity, this dependency is exclusive. The components are stimulated only through their I/O. In Parallel DEVS, parallelism is exploited when possible by providing a means by which all imminent components can execute concurrently in any given simulation cycle.

Concurrent flows with single or multiple servers among them can provide a concrete basis to a wide range of semantics for control as well as data flows among the

activities. Useful simulations can be obtained to observe and monitor different behaviors of models that have been attained incrementally through complementary views of their representations at higher layers. The granularity of the time base may vary depending on the system and the input/output requirement. For example, the behavior can be observed for a limited time base for some purposes among which testing or debugging for specific scenarios can take place. Evaluation of the performance is also essential and can take place through crafted simulations with specific observation capabilities.

9.3 The Specification of Action and Control in Activities

The activities metamodel (OMG, 2012) essentially consists of the major flow and node elements. The flow can be classified into control or data flow types. The node can be classified further in many different types. Action is a major node in activities. It is a fundamental unit in dictating the behavior of systems. Along with other types of nodes, such as control nodes, they form the overall behavior in an encompassing activity. There are four major types of control nodes, *decision*, *merge*, *split* and *join* nodes. Each one has some semantics to handle flows in activities.

Action is an abstract classifier to represent a wide range of possibilities for the model to behave in many different ways. In (OMG, 2013), several types are introduced to provide a means to handle different behavioral aspects. For example, structural feature actions manipulate structural features, including reading and writing. We conceive the notion of action based on system-theoretic principles, particularly the DEVS formalism. Therefore, it needs to align with the notion of the component as well as state as defined in DEVS. Action is not merely an intermediary means upon state transitions. In (Shoham, 1989), action defines state. In the context of activities, action, and control elements are both specializations of the node type. They

can be used to represent different dynamics at a higher level before realizing their implementations. In previous work (Alshareef and Sarjoughian, 2018b), we devised an interpretation facility to support the simulation of activities in the DEVS-Suite simulator (ACIMS, 2019).

The control nodes provide a suitable means to direct the flow, including the ones carrying data. The coordination task can be specified using different variations of controlling constructs based on the dynamics of the system of interest. For example, coordinating for a conventional multi-server system requires directing jobs to different servers. It could also require maintaining the status of each server. The coordinator could also maintain the queue. The state variables of such a system are discussed in Section 9.2. The controlling constructs of activities can be selected to be a consistent representation of the coordinator when possible considering the semantics of each construct. The *decision* node is used to determine which outgoing flow is selected. The *merge* node could receive multiple flows. As soon as it receives a flow, it directs it to the corresponding node. The *split* node is used to produce multiple flows concurrently while the *join* is used to synchronize multiple incoming flows. These constructs and different combinations of them can collectively serve as a representation for the desired coordination dynamics. *These are significant relative to specifying behavior solely in terms of states and transitions with strong simplifications on actions.*

Actions of activities can have input and output pins. Unlike actions, control nodes do not have such means. Therefore, as a semi-formal approach, it needs to be compensated with such a definition of the port as a necessary mechanism for handling the I/O. In the DEVS formalism, communication among different components can only exhibit through the designated ports. In our approach, the action is encapsulated within the component and yet explicitly specified to define the total behavior. The multiplicity may vary based upon the complexity of the behavior of which the activity

constructs are set to represent (see Figure 9.1a). The activity itself can have input and output parameters. Both parameter and pin are specializations of the object node. They both serve as means for communications among and between activities. The former is for communicating between activities, and the latter is for communicating among them.

Thus, we create an activity of the multi-server archetype architecture (Figure 9.3). The activity receives its input through *input parameter*. The coordination decision node then decides based on state variables and depicted in the conditions C_1 and C_2 to which server the job is to assign. The subsequent splitting nodes then produce two flows, one to the server, and another for $Merge_1$ node. The flow to $Merge_1$ is devised to notify *output_t parameter* about the job assignment to server. This notification can be used for experimental purposes such as performance evaluation. After servicing the job, a flow is issued from the server to $Merge_2$ which is then directed to the *split* node. The latter node produces two flows, one to *output parameter*, and another to notify the coordinating procedure about the availability status of the corresponding server.

For richer model specification, the semantics of the described connections in this approach is complemented by the semantics of couplings in Figure 9.1a and the semantics of flows in Figure 9.3. On the one hand, the flows are endowed with the formal and precise semantics of coupling. On the other hand, different types of flow at different layers provide the means for richer specification with the notions of control and data. Having such a capability is significant with proper use of model-driven methodologies across abstraction layers. The elements within the dashed line area in Figure 9.3 show the presence of two internal and one external output couplings for the coordinator component.

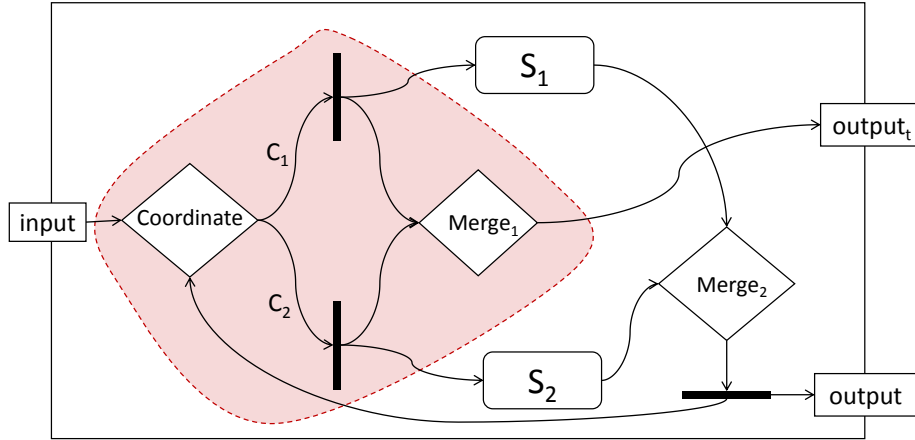


Figure 9.3: Activity of a Multi-Sever Archetype Architecture Is Devised Using Various Activity Constructs. S_1 and S_2 Actions Represent the Jobs Service. C_1 and C_2 Represent Conditions for Choosing Flow Directions. The Nodes inside the Dashed Line Area Highlight the Role of the Activity Control Elements in the Manipulation of the I/O Flow.

9.3.1 Coordinator Statecharts

We devise the coordinator as a component in Figure 9.1 and therefore, its behavior can be looked into from the statecharts standpoint. The statecharts (see Figure 9.4) mainly consists of two states, *passive* and *sending*. The passive state is to represent the state of the coordinator when it is not manipulating any job. When receiving an input, then the coordinator checks through which input port the input has been received. Input through *in* indicates arrival of a new job. The coordinator fetches the job and figures out its service time. It iterates through the secondary state variable that maintains the availability status of servers. If there is an available server, the job gets assigned to it, and a transition to *sending* state occurs. If there is no available server, then the server remains at *passive* state and the job gets stored in the FIFO queue. The inputs through port *notify* indicate the completion of job servicing and

consequently, the availability of the designated server. Therefore, a transition to *sending* occurs if the queue has some jobs waiting for service. The job is removed from the queue and dispatched to the available server. If there is no waiting job in the queue, then no transition is made, and the model stays at the *passive* state.

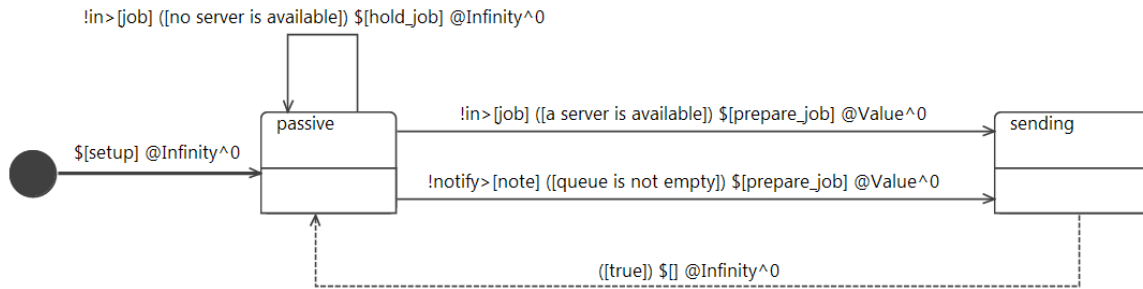


Figure 9.4: Modeling the Coordinator Statecharts in CoSMoS.

Both activities and statecharts are approached from a complementary standpoint. Together they provide a richer basis to specify behaviors, although challenges may arise to establish a total perspective by which the notion of action and state are both fully considered. Neither can alone dictate the overall process of specifying behavior nor eliminate the necessity of manipulating codes to provide proper implementations.

9.3.2 Constructing Hierarchy within Activities

Based on the DEVS hierarchical model specification, an activity model can conceptualize as a layer in a hierarchy. Such hierarchy leads to activity models at higher abstractions using activity model elements. For example, a control node has a higher abstraction relative to the primitive fork node. Higher-level elements represent higher-level concepts as compared with primitive elements. Thus, activity models at multiple abstraction levels can be constructed. In other words, higher-level activity models can place constraints on the primitive elements used in activity models. Given the lack of hierarchy concept directly in the standard UML activity modeling, we consider

using the activity model elements according to the DEVS model hierarchy and its corresponding abstract simulator. Constructing and coupling activity models for the atomic DEVS models lead to specifying coupled DEVS activity models that conform to the DEVS closure under coupling property. Behaviors of certain activity elements can be determined using their DEVS specifications. For example, a part of an external transition function can be defined using a fork node and a part of an output function can be defined as a decision node. Together, these activity elements can define an internal coupling between two atomic models contained in a coupled model. This approach supports defining hierarchical (higher-level) behavior at finer-grain abstraction levels. Such support is useful as behavior specification for hierarchical components can have additional details.

The model in Figure 9.3 has a flat structure (i.e., there is one coupled model which contains several atomic models). The hierarchy level for this model is one. Now, we redefine this model by replacing both of its atomic models that correspond to the servers S_1 and S_2 with two coupled models. In this hierarchical example, the model hierarchy level is limited to two. The contained coupled model, which is created to correspond to the second server, does not contain any coupled models. This example shows the communication semantics between activity models according to the DEVS formalism as well as the semantics of the activity model elements. The aim is to help better understand and specify behaviors for coupled models using activity modeling. The DEVS simulation protocol provides the semantics for executing activities within atomic models as well as the input and the output communications between models. Communication between two activities belonging to different models is defined between any main activity and its nodes.

The activity node S_1 is reconsidered in Figure 9.5 and replaced with a distinct activity (*Activity 1*) that is being communicated with through signals sent and re-

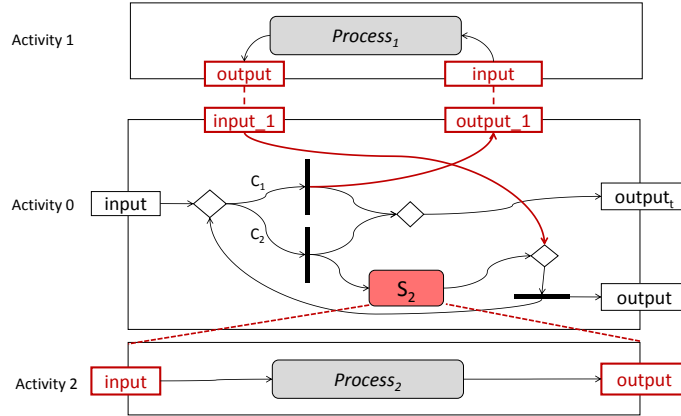


Figure 9.5: Hierarchical Construction with Activities.

ceived from the main *Activity 0*. The node S_2 becomes also an activity (*Activity 2*), but it is contained within *Activity 0*. Elements in red (gray) color represents the model elements added to satisfy DEVS modularity. These two cases have different representations and hierarchy specifications in DEVS. The former leads to a coupled model at a hierarchical level corresponding to the main activity. The coupled models communicate through ports and external I/O couplings. The latter leads to establishing a new hierarchical level with the coupled model at hierarchy level one contained within the coupled model at hierarchy level two. Figure 9.5 illustrates how the two different cases can represent in correspondence to the main activity.

9.4 Demonstrating with Activity Modeling Tool

The development of the tool starts by creating an Ecore model to account for the activity metamodel to support the creation of activity models in a hierarchical fashion. The hierarchy is accounted for in the composition relationship (see Figure 9.6) between *Behavior* and *Activity* elements. The Ecore model also shows the other *EClass*, *EReference*, and *EAttribute* elements that are used to facilitate the model creation process.

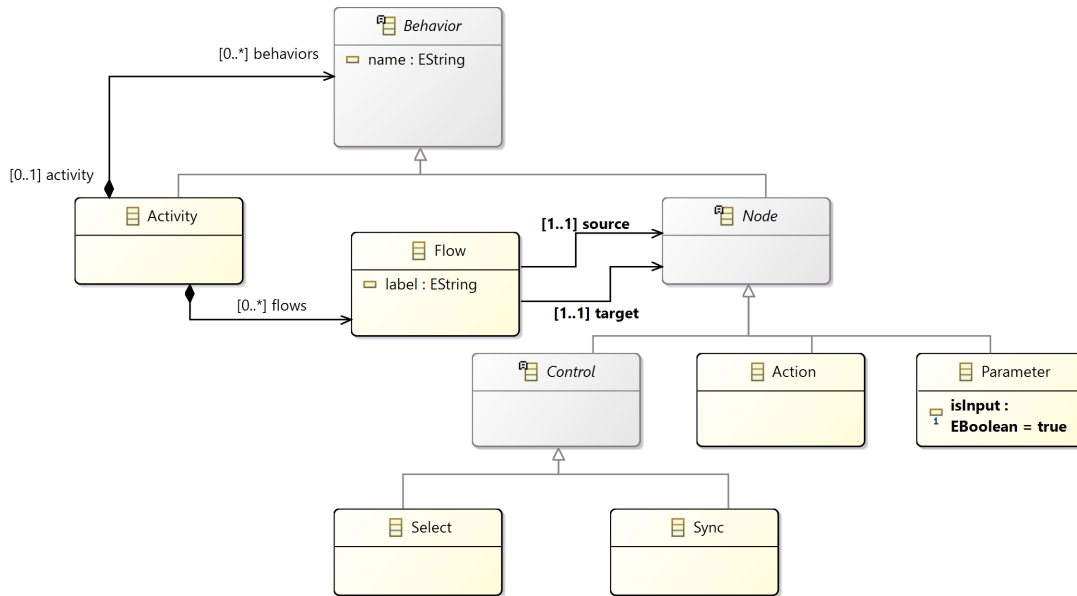
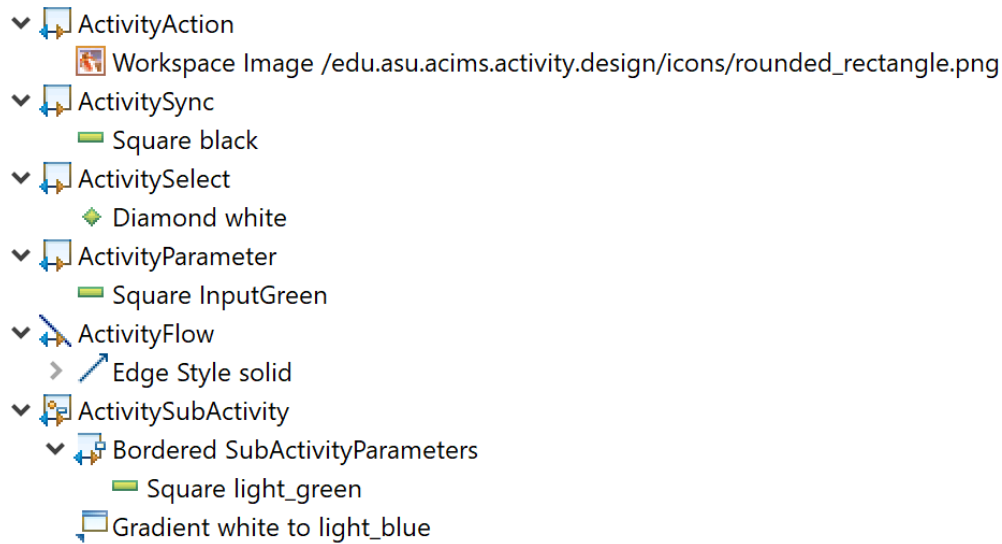
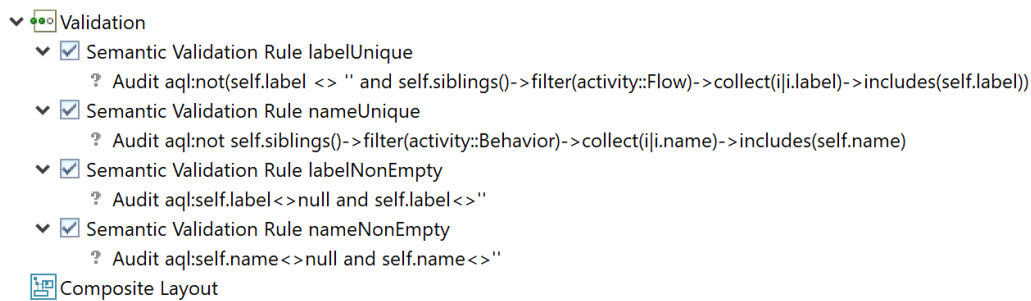


Figure 9.6: A Metamodel for Hierarchical Activities Developed Using Ecore.

The graphical properties are defined using Viewpoint Specification in Sirius (Eclipse Foundation, 2018). Figure 9.7 shows some screen-shots of different parts of the specification. Figure 9.7a shows the part for defining the geometric shapes by which the model artifacts are to visualize. For example, a rectangle with rounded corners refers to the action node. Similarly, other shapes are defined for other nodes in addition to the edge to represent activity flow. Figure 9.7b defines some rules to issue a warning or error message to notify the modelers. Examples of rules are shown in the screenshot, such as the label name's uniqueness and setting. Figure 9.7c shows the section parts where geometric shapes are associated with their counterpart definition from the domain model (Ecore). The section offers further capabilities to manipulate other properties such as the context of the created element. It also offers to modify the instance model upon certain changes. The rules need to be defined within the viewpoint specification file using languages such as Object Constraint Language (OCL) or Aceleo Query Language (AQL). An example is the deletion of flows upon the



(a) The Part of the Viewpoint for Different Nodes Creation with Their Designated Geometrical Shapes.

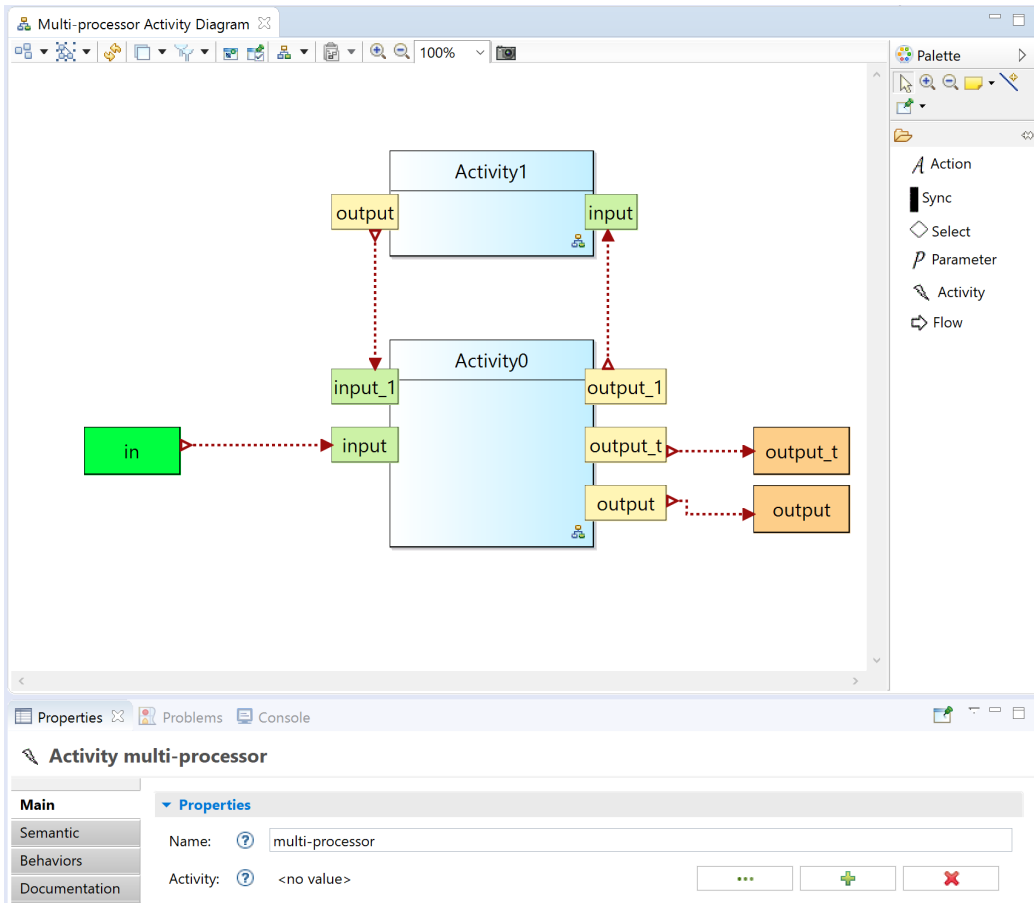


(b) The Part of the Viewpoint for Validation Such as Ensuring Label Name Uniqueness and Setting.

Figure 9.7: Viewpoint Specification in Sirius.

deletion of their corresponding nodes, which is shown in Figure 9.7c.

The tool then can be used to create diagrams such as the one for the multi-server activity (Figure 9.8). Figure 9.8a shows the creation of the main activity along with the tool palette. In the subsequent Figure 9.8b, 9.8c, and 9.8d we only show a screenshot of the canvas. After developing these activities, the code generation process can take place. The code generators are implemented using Acceleo and the result in the



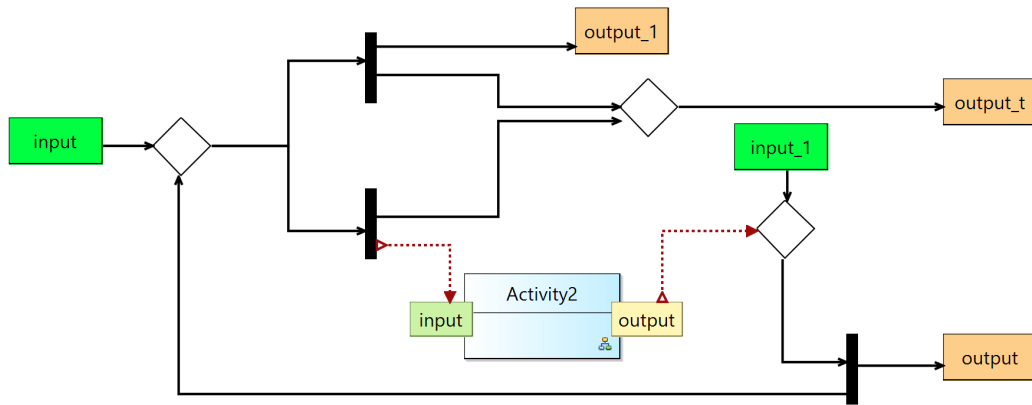
(a) The Screen-Shot Shows Developing the Main Activity Diagram for the Multi-Server with Hierarchical Construction as Described in Figure 9.5. In Addition to the Canvas, the Palette and the Properties View for the Activity Are Shown.

Figure 9.8: Modeling Multi-Server Activity in the Developed Activity Modeling Tool.

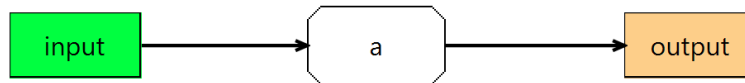
set of models as java files that are necessary for the simulation to take place in the DEVS-Suite simulator, as shown in Figure 9.9.

9.5 Related Work

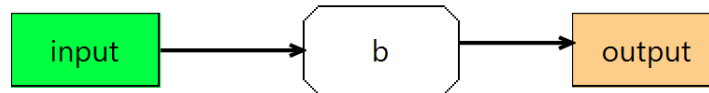
Enriching models at the higher levels of abstractions while providing supports for implementations via different code generation facilities has been of interest for



(b) The Corresponding Diagram for Activity 0.



(c) The Corresponding Diagram for Activity 1.



(d) The Corresponding Diagram for Activity 2.

Figure 9.8: Modeling Multi-Server Activity in the Developed Activity Modeling Tool.

many research efforts (e.g., Lei *et al.* (2009); Cetinkaya *et al.* (2012); Sarjoughian and Markid (2012); Kapos *et al.* (2014); Mittal and Martín (2013a); Sarjoughian *et al.* (2015)). These works address the problem holistically and yet thoroughly for the specifications in general (Lei *et al.*, 2009; Cetinkaya *et al.*, 2012; Mittal and Martín, 2013a; Kapos *et al.*, 2014) and for behavioral specifications particularly (Sarjoughian *et al.*, 2015). In Lei *et al.* (2009) MDA is serving the aim of model transformation and primarily from a structural vantage point. In (Cetinkaya *et al.*, 2012), structural metamodeling is the focus. In (Mittal and Martín, 2013a), the authors attempt to integrate the concepts in Model-Based System Engineering (MBSE) and Model-Driven

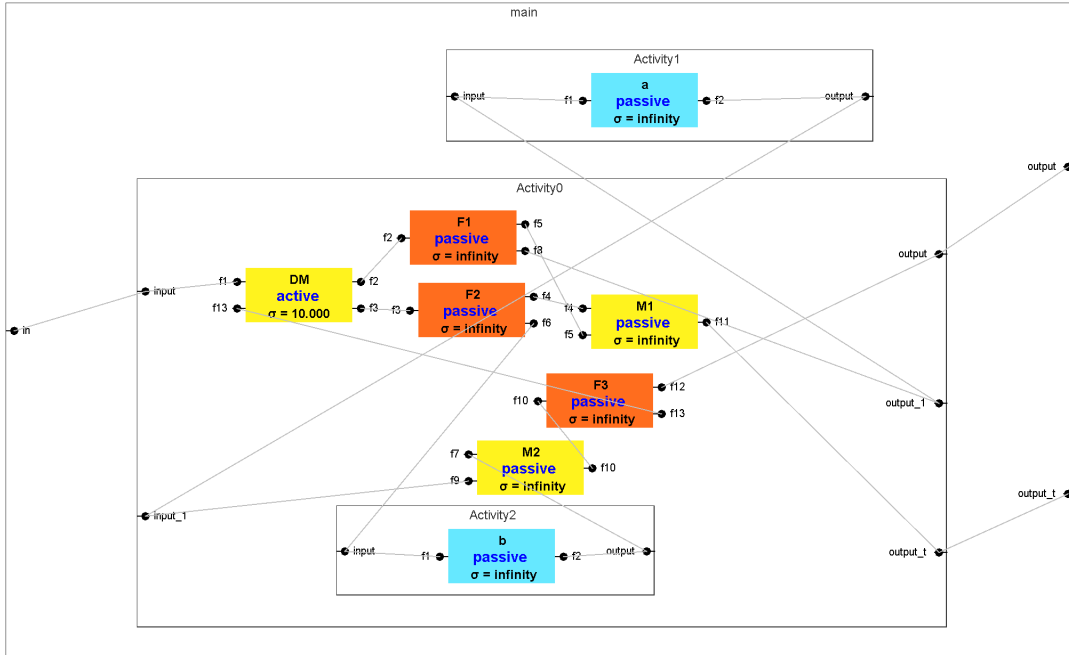


Figure 9.9: The Simulation View of the Developed Activity for the Multi-Server System after the Code Generation for DEVS-Suite Simulator.

Engineering (MDE) for the DEVS by employing modeling frameworks and tools that are MDA-based. The notion of activities is not explicit. In Kapos *et al.* (2014), the integration is for the System Modeling Language (SysML) based on the similarities between DEVS and SysML. Therefore, SysML definitions are carefully examined relative to DEVS concepts. A subset of SysML aligns with DEVS by conforming to the SysML profile devised in close consideration to DEVS concepts. Models of such characteristics in the devised profile are made simulatable in a DEVS environment via transformation and code generation. In (Sarjoughian *et al.*, 2015), the extension mechanism is accomplished by defining behavioral constructs and then incorporating them into the DEVS metamodel itself. It describes the action in the metamodel and associates it with different state transitions through meta-behavior based on EMF-DEVS (Sarjoughian and Markid, 2012). The DEVS metamodel extends the Ecore

metamodel of the Eclipse Modeling Framework (EMF). Kapos *et al.* (2014) used activities of SysML to define parts of the atomic model behavior along with other state and parameter diagrams. The work targets SysML at large, including its various metamodels such as those of block, state, parameter, constraint, and activity. The total SysML model is made simulatable after adequately providing the specification required by a DEVS simulator. All the mentioned works comply to varying degrees with the MDA hierarchy to support the creation of models that conform to their corresponding metamodels in a disciplined incremental manner.

This work focuses on notions of activities, mainly control and action. We propose an explicit definition of action and control in the DEVS formalism across different modeling layers. We also examine a relation where, on one end, is the hierarchical component models and on the other end, resides action and control as fundamental units for identifying behaviors. As a result, richer activity models can be obtained. The notions of component and state are both used according to their underlying system-theoretic concepts. The action is introduced based on its definition such as in the UML (OMG, 2012) and with consideration of existing DEVS variants that explicitly define it such as ALRT-DEVS (Sarjoughian and Gholami, 2015). These definitions align across meta-layers, and their corresponding implementations in CoS-MoS are examined.

Harel and Politi (1998) implemented a system with different capabilities, among which the ability to simulate one-step or in an interactive manner. The actions are defined to be carried out only instantaneously during a transition between different states. The formalism is set for the system under development (SUD) to provide a means for such a system to be developed by multiple modelers from different standpoints. The statecharts govern the activity and actions. Also, the component is defined for the physical module. In the DEVS formalism, the time advance function

is defined for the atomic model. It could be assigned zero in the case of zero time advance. However, successive zero time state transitions must be finite to guarantee model legitimacy. Starting in the 1990s a variety of efforts began on using DEVS and various kinds of state machines as complementary paradigms, but they did not consider activity modeling.

In (Störrle *et al.*, 2005), the authors consider Petri nets for ascribing semantics to activities in UML 2.0. The work considers the basic Petri nets formalism along with several extensions of it to address further some more expressive properties in activities such as control and data flow. Authors use extensions such as Colored Petri nets, Procedural Petri nets, and extension of the Procedural Petri nets. These extensions are used to try to incur further expressiveness encountered in activities but not in Petri nets. They conclude that mapping activities to Petri nets do not scale, meaning that extending the mapping with relatively non-trivial behavioral aspects thereby breaks the basic intuitions. Although limited, such transformations can lead to verification of specific properties in the context of the component-based model.

9.6 Conclusion

Some capabilities, such as parallel processing and synchronization, are exploited in the proposed modeling approach with the exemplar dual-server model. These capabilities have been possible using the underlying DEVS formalism and defining models in correspondence with activity-based behavior specification. Such models can aid modelers to confine and develop simulations based on their well-defined semantics, both structurally and behaviorally. The proposed approach suggests the use of more intuitive diagrams to facilitate the development of models having complex dynamics. Hence, some of the described capabilities are not obvious due to using either the semi-formal or formal modeling methods.

The use of multiple abstractions exhibits the difficulty of handling ambiguities in some prominent modeling languages and terminologies. Rough definitions may accumulate, but they may increase the burden placed on the shoulders of modelers, especially when the aim is to arrive at more useful simulations. Accounting for different abstractions can strengthen the process of incremental model development. Modelers are obliged to raise fundamental questions about their models, starting from basic concepts. As a result, the discovery of such models and their corresponding suitable abstractions (MDA four-layer architecture) is guided but remains in part unrestricted. The correctness of multiple entities in a given modeling framework is employed to increase the rigor needed for systems that continue to grow in both scale and complexity.

ACTIVITY SPECIFICATION FOR TIME-BASED DISCRETE EVENT
SIMULATION MODELS

Dealing with different parts of a system model can be problematic particularly when they are scattered within and across different abstraction hierarchies. At some point, the abstraction layers of a hierarchy, each possibly having multiple levels within, have to be bounded with some constraints to make them useful and prevent issues such as circularity relationships. For example, the model-driven architecture (MDA) is defined as a four-layer hierarchy from M0 through M3, where the former represents the most concrete, and the latter represents the most abstract. Although useful, it remains challenging to define boundaries and relationships in a clear-cut manner, for example, from the standpoint of executable models and simulation in particular.

The problem of partitioning models into components and relationships becomes evident for structure as well as behavior. For the structure, complications may arise with a significant increase of multiple message types and communications requiring computation synchronization and concurrency. For the behavior, dissecting the internals of communicating components of a system can also pose difficulties in developing executable abstractions. At some point, nonetheless, behavioral specifications at multiple levels of abstraction must take place in controlling and conducting some lower level computing tasks. The delegation of lower level tasks becomes challenging due to the central role abstraction hierarchies play in managing complexity and scale across complementary model specifications.

Considering the ongoing efforts in further deepening the hierarchy for component-based models, we propose examining and using the action and control elements at

the meta-layer activities, mainly focused on the M1 and M2 layers. Therefore, the notions of component and state are both used in activity modeling based on their underlying system-theoretic concepts. Moreover, the notions of action and control are also used to complement component and state definitions. Our goal is to arrive at an understanding of the different roles played by each part, especially from multiple meta-layers viewpoints. Moreover, the specifications of such layers are distinct and explicit for simulation modeling purposes.

In discrete systems (Wymore, 1993; Zeigler *et al.*, 2018b; Alur, 2015), a variety of needs institute the time scale upon which the system representation or approximation can execute. Real systems such as smart manufacturing or transportation are known to be large in both scale and complexity. Such systems have numerous components with various types of connections between them. Components do not have to resemble each other except at high levels of abstraction that are often rendered difficult to concretize. Given these types of systems, some models, such as synchronous reactive components, provide strong constraints in terms of timing, state changes, and composition.

In contrast, discrete models can be asynchronous. For a complex discrete system, simpler models with single-input and single-output can be placed within architectures that pose different multiplicities in terms of their inputs and outputs. Observing the degrees of detail with sufficient confidence is challenging, yet essential, in order to proceed in the process of the system as well as simulation development.

The Turing machine (TM) can be described as a discrete system, and the Discrete Event System Specification (DEVS) describes TM as a modular composition according to the hierarchy of system specifications (Zeigler *et al.*, 2018b). Interesting results can disperse via simulation studies; however, sophisticated observations can be accessible and clearly understood only through rigorous experiments. Useful analyses

such as throughput might be achievable through making such models subject to full-fledged experimental designs, but such analyses with a keen sensitivity to temporal aspects are impossible or otherwise hard to achieve.

In this chapter, we examine activity specification as a standalone approach across different modeling meta-layers with the intent to arrive at useful simulatable models. We first discuss some frameworks and architectures that can facilitate modeling. Then, we present the DEVS specification for activity modeling. In the remaining sections, we detail the semantics of the modeling approach in conjunction with demonstrations of certain aspects of multiprocessing architectures.

10.1 On Simulation Modeling Architectures and Frameworks

We begin with discussions about related works and background regarding the development of architectures and frameworks to support discrete event simulation modeling. First, we give a brief description of MDA and then discuss some of its concepts, particularly when applied to modeling and simulation. Second, we highlight a few existing studies and present the researchers' viewpoints regarding what accounts for the ongoing efforts in dealing with models that can be developed using different abstraction means.

10.1.1 Modeling Layers

An earlier study, Sarjoughian *et al.* (2015) proposed a metamodel for the DEVS atomic model spanning a variety of concepts and techniques based on MDA. We extend the core Eclipse Modeling Framework (EMF; Steinberg *et al.* 2008) model with primitive notions for behavioral specifications in an attempt to make behavioral modeling possible along with structural specifications. Although useful, there are some inherent limitations of using such means for behavioral modeling.

The MDA layers M3, M2, M1, and M0 lay the groundwork and guidelines for incrementally developing models of component-based systems. These guidelines can be useful if followed carefully. However, the nature of extending techniques among metamodels may result in mere complications with unnecessary complexity and overhead (Fondement *et al.*, 2013). Concepts at a meta-layer necessitate further efforts to substantiate them, which can often be expensive. Across all the MDA layers, the key idea is to create a classifier and multiple extensions and instances thereof. The directions of extension and instances are believed to be orthogonal. However, the Object Management Group (OMG) has made more elaborate standards by which distinctions of notions of cross meta-layers and within a single meta-layer are drawn. Interpretation and the latter by instances ascribe the former. In some cases, extensions reside horizontally within the same meta-layer in the hierarchy, and instances reside vertically in the next layer below. The connection is reversed in other cases. In our work, we observe both standards and attempt to deal with the subtlety of the issue by relying on the theory of modeling and simulation for a demonstration from a system-theoretic vantage point.

It is essential to establish a more rigorous means of facilitating the creation of models at a concrete layer. It is also significant and yet far more challenging to realize, with confidence, connections between models at the concrete layer and their counterpart abstractions at some high layer. The difficulties symmetrically increase with layers that are higher in the hierarchy. We thoroughly examined the concepts and presented works that attempted to map concepts from upper layers downward using a variety of methods. The results are promising for relatively simple systems but have generally proven to not be particularly useful for complex systems. We will discuss this further in the following section.

10.1.2 Related Work

Leaping models and abstractions of models to some counterpart manifestation at a concrete layer is increasingly recognized as a significant problem that has been a topic of great interest to a number of researchers. Some researchers have focused primarily on efforts to realize implementations of models based on model-driven and model-based support for engineering. The system-theoretic standpoint varies significantly among these efforts. In statecharts (Harel and Politi, 1998), models are perceived to construct the system under development. In some others, the system viewpoint is absent, and yet the effort is focused on ascribing semantics for the intended modeling language (Störrle *et al.*, 2005). While many others make a more deliberate attempt toward employment of model-driven frameworks to integrate simulation as a means for precisely observing the system under study (Yonglin *et al.*, 2009; Risco-Martín *et al.*, 2009; Kapos *et al.*, 2014; Bocciarelli *et al.*, 2019).

Efforts have been made at utilizing MDA to provide model transformation frameworks. A general objective in certain studies (Yonglin *et al.*, 2009; Risco-Martín *et al.*, 2009) was to promote model reuse across different platforms. It is quite often the case that specific capabilities are offered in a target platform but do not apply in others, which led to the notion of platform-independent solutions. The problem may also become more difficult for inherent considerations. For example, different timing accounts pose key challenges across execution environments. The challenge can grow significantly when transforming between time agnostic means to environments with stronger accounts for various temporal aspects and techniques. Timing is a significant issue in crossing between different modeling environments by which many of the proposed transformation models between UML and DEVS have been affected. MDA alone falls short of providing a concrete solution for behavioral specification, although

it has been essential in guiding certain advances in different DEVS-based modeling frameworks (e.g. Yonglin *et al.*, 2009; Foures *et al.*, 2012; Risco-Martín *et al.*, 2009).

On the one hand, efforts and standards have pushed toward enabling the creation of a platform-independent model (Soley and the OMG Staff Strategy Group, 2000; OMG, 2017). On the other hand, the feasibility of executing these models with techniques such as code generation becomes subject to fundamental questions with knowledge gaps and arbitrary semantics (Nikolaidou *et al.*, 2016; Zeigler *et al.*, 2018a; Alshareef and Sarjoughian, 2018b). Profound simulations become of particular importance when it comes to realizing behavioral specifications of different models. As such, this paper can be characterized as an attempt to achieve that goal.

In previous work, Alshareef and Sarjoughian (2017), Alshareef *et al.* (2018), and Alshareef and Sarjoughian (2018b) laid the groundwork for the modeling and simulation of activities regarding parallel DEVS formalism and its abstract simulator. Some exemplary models demonstrate the basic mapping of action to the atomic model, as proposed by Alshareef and Sarjoughian (2017). The mapping attempts to utilize the full capability of a rich simulation framework as opposed to debugging (Mayerhofer, 2012) or execution with a fixed time step. We discussed the approach and the mapping in more detail (Alshareef *et al.*, 2018) with regard to the I/O function in the system specification hierarchy. Recently, Alshareef and Sarjoughian (2019) extended the work to the coupled component with a focus on the model hierarchy to facilitate the construction of the component-based models. Here, we propose DEVS models to account for the syntax and semantics of the control nodes of activities. We also propose a framework for simulating activity models while characterizing their object and control constructs and examining their usage that accounts for different timing considerations. In doing so, we attempt to make use of the experimental frame to acquire richer analyses of activities across different abstraction layers.

10.2 DEVS Specifications for Activity Nodes

Previous studies (Alshareef and Sarjoughian, 2018b; Alshareef *et al.*, 2018) have examined different semantics of activities and created a set of specifications that correspond to various elements in the UML activity metamodel. We formulate the specifications primarily for two types of activity elements known as action nodes and control nodes.

In a nutshell, every activity is essentially a graph that consists of nodes and edges. Edges are referred to as flows, while the nodes can be an object, control, or action. Control nodes include *join*, *fork*, *merge*, or *decision*. The *fork* node is the one that synchronizes the production of outputs through its outgoing flows. Similarly, *join* synchronizes the flows but regarding its inputs where it expects an input through each incoming flow. Because they are symmetric, we will later refer to *fork* and *join* together as the *SYNC* specification (Listing 10.1). In the same vein, the *merge* and *decision* nodes are used to select one flow for proceeding. In the former, it is incoming, and in the latter, it is outgoing. We will refer to them jointly in the *SELECT* specification (Listing 10.2).

Action nodes are the most fundamental unit of behavior in the Unified Modeling Language (UML) 2.5 metamodel (OMG, 2017). They are defined as an abstract node in the metamodel and are refined in the foundational subset of the executable UML Models (fUML) (OMG, 2018). Sets of specific actions are sub-types of the abstract action. For example, one category suggests a collection of actions to be reading actions, and therefore, their specifications are partially defined in the UML specification. To provide an implementation for such descriptions, the standard provides a mapping to the Java programming language through interpretations. The capability of running fUML models is delivered through model execution environments such as

Moka within Papyrus (Eclipse Foundation, 2016b). The modeling capability is in Papyrus. In our work, we focus on providing an execution capability by exploiting a DEVS-compliant simulator such as the DEVS-Suite simulator (ACIMS, 2019). We argue that the inherent account of time (i.e., as a standalone part of an executable model) is necessary, to an extent, to navigate through the semantics of various activity constructs (Alshareef and Sarjoughian, 2018b).

Control nodes in activities are also essential for guiding the flow. Their roles varies; however, we mainly categorize them into two major types. The first type consists of the *fork* and *join* nodes. The second type consists of the *decision* and *merge* nodes. We refer to the first type as *SYNC* and the second type as *SELECT*. These two major types mainly differ from each other in the synchronization of their incoming and outgoing flows. In the former type, the flows are synchronized, but that is not necessary for the latter. The specifications of different nodes are discussed in previous studies by Alshareef and Sarjoughian (2017); Alshareef *et al.* (2018); Alshareef and Sarjoughian (2018b).

Next, we define a formalized mathematical specification for each type according to the parallel DEVS formalism. Listing 10.1 shows a formal specification for the first type. The syntax and semantics of this specification, as with the second type, strictly conform to the parallel atomic DEVS model abstraction.

Listing 10.1: *SYNC* Atomic DEVS Model Specification

$SYNC = \langle X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$, where

$$X^b = \{(p, v) : p \in IPorts, v \in X_p,$$

$$IPorts = \{in_1, \dots, in_n\}, X_p = V(\text{an arbitrary set})\}$$

is the set of input port and value pairs;

$$Y^b = \{(p, v) : p \in OPorts, v \in Y_p\},$$

$$OPorts = \{out_1, \dots, out_m\}, Y_p = V(\text{an arbitrary set})\}$$

is the set of output port and value pairs;

$$S = phase \times \sigma \times task \times C, \text{ where}$$

$$phase = \{passive, waiting\}, \sigma = \mathbb{R}_{0,\infty}^+, task \subseteq X^b,$$

$$C = \{(p, c) : p \in IPorts, c \in \{true, false\}\}$$

is the set of input ports and conditions;

$$\delta_{ext}((phase, \sigma, C, task), e, X^b) =$$

$$(waiting, \infty, (p_i, true), (p_i, v_i))$$

$$\text{if } p_i = in_i \wedge \exists (p_j, c_j) \ni c_j = false, i \neq j$$

$$(waiting, 0, C, (p_i, v_i))$$

$$\text{if } p_i = in_i \wedge \forall (p_j, c_j) \ni c_j = true, i \neq j;$$

$$\delta_{int}(phase, \sigma, C, task) =$$

$$(passive, \infty, nil, x) \quad x \in X^b;$$

$$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$$

$$\lambda(waiting, \sigma, task) = (p, task);$$

$$ta(phase, \sigma) = \sigma.$$

In the case of *join*, the node expects the arrival of input via all incoming flows before dispatching output. Therefore, this is represented in the *SYNC* specification by having multiple input ports. The state is used to distinguish incoming inputs arriving on multiple ports from one another. The distinction is carried out via $(p, c) \in C$. As soon as all expected inputs have arrived, the output is dispatched with a zero time advance assuming no delay is expected to take place for producing and dispatching the output. The *fork* behaves similarly, except that multiple outputs follow for a given input.

The formal specification corresponding the second type of control node (for *merge* and *decision*) is detailed in Listing 10.2.

Listing 10.2: *SELECT* Atomic DEVS Model Specification

$SELECT = \langle X^b, Y^b, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$, where

$$X^b = \{(p, v) : p \in IPorts, v \in X_p,$$

$$IPorts = \{in_1, \dots, in_n\}, X_p = V(\text{an arbitrary set})\}$$

is the set of input ports and values;

$$Y^b = \{(p, v) : p \in OPorts, v \in Y_p,$$

$$OPorts = \{out_1, \dots, out_m\}, Y_p = V(\text{an arbitrary set})\}$$

is the set of output ports and values;

$S = phase \times \sigma \times task \times C$, where

$$phase = \{passive, sending\}, \sigma = \mathbb{R}_{0, \infty}^+, task \subseteq X^b$$

$$C = \{(p, c) : p \in OPorts \text{ and } c \in \{true, false\}\}$$

is the set of output ports and conditions;

$$\delta_{ext}((phase, \sigma, C, task), e, X^b) =$$

$$(sending, 0, C, (in, v_1), \dots, (in, v_n));$$

$$\delta_{int}(phase, \sigma, C, task) = (passive, \infty, (p, false), task);$$

$$\delta_{con}(s, ta(s), x) = \delta_{ext}(\delta_{int}(s), 0, x);$$

$$\lambda(sending, \sigma, C, task) =$$

$$(p_i, task) \text{ if } \exists (p_i, c_i) \ni c_i = true;$$

$$ta(phase, \sigma) = \sigma.$$

The above specification is generalized for the *decision* and *merge* nodes. We note that elaboration has to take place to account for execution semantics. The structural part of this specification (input, output, and state constructs) is the same as for the *SYNC* model. The behavioral specification represents the dynamics of the

SELECT model. The *SYNC* and *SELECT* models have the same time advance function specification.

In previous reserach, Alshareef and Sarjoughian (2017) discussed in detail a discipline for a network switch example. The selection specification describes the decision node where the flow is directed in such a network based on a polarity condition. The condition is maintained and updated at each input arrival time. The *SELECT* specification resembles multiple aspects of the example in the previous work. We will discuss this specification with a slightly more concrete example in the following sections.

10.2.1 Mapping UML Activity Control, Object, and Flow to DEVS Model, Port, and Coupling

We note that according to the activity metamodel in the UML, the notion of a pin can be defined only for the executable activity node. The pin can be an input, an output, or a value pin. Other types of activity nodes (e.g., control node) cannot be defined with pins, and therefore, the handling of I/O is tacit or left undefined. In the DEVS formalism, an atomic model can be equipped with a finite but unrestricted number of ports. Each port, whether it is input or output, can be arbitrarily attached to one or more internal or external coupling. Thus, we create two ports for each channel (i.e., coupling), one as an output and the other as an input, where the coupling is added to link the two ports. A channel, with its input and output ports, corresponds to a flow in the activity diagram. It serves as a means to transfer I/O through different models, whether they correspond to actions or control nodes. By assigning a distinct port to each coupling, we eliminate the possibility of duplicating I/O in the DEVS network and therefore needing some elaborate mechanism for handling many ports per coupling. Each I/O or some part thereof gets transferred only to the intended

element.

From the UML vantage point, flow is classified by control and object flow types, each having its syntax and semantics. A closer examination of the flow types for the fUML model reveals the notion of *locus* to facilitate execution by carrying out (i.e., transmitting) information through activity nodes during the execution life cycle. Control flow is defined to dictate order, while object flow is defined to also dictate order but to do so while carrying data between different nodes. Control nodes do not distinguish between flow types, nor do they require pins to link with object flows. The pins are designated for carrying whichever kind of flow they are connected with. They pass any received object along to its designated nodes without any manipulation. An action can have, at most, one input pin and, at most, one output pin. An action node's input and output pins can be linked to object flows; other flows for the action node can be of a control type. Action can receive and produce as many object and control flows as possible, but a finite number of flows can link to action, whether incoming or outgoing. Control nodes cannot connect with pins, but they can relate with as many flows as possible, whether object or control. In our proposed approach, we ignore this classification, and every flow is defined as coupling. That is, we make no distinction between object and flow controls. We account for the syntax mentioned above and semantics through the definitions of the port of atomic/coupled DEVS models. In addition, the arbitrary handling of I/O is due to the DEVS abstract execution protocol, which is domain agnostic.

SYNC activity model:

In this atomic model (Listing 10.1), one input port is defined as corresponding to each incoming flow. An output port is also designated for each outgoing flow. For example, if the *join* node has two incoming flows and one outgoing flow, then the correspondent

atomic model would have two input ports and one output port. Then, a coupling is attached to each port. Therefore, the *SYNC* atomic model initializes in a passive state. As soon as it receives input through one of its input ports, it transitions to a waiting state until other incoming inputs arrive from different ports, and the output does not dispatch until all required input arrive. When an input arrives through each input port, then the model transitions to a different state, after which it combines all inputs and prepares the resulting outputs. The combining procedure is absent from the metamodel of activities (i.e., the specifics of a procedure are to reside in a concrete model according to some given application domain). The output then dispatches through all output ports, and therefore, their distinctive couplings are used for delivery to their destinations.

The SYNC model describes the correspondence to both the *join* and the *fork* nodes, and thus, it accounts for the syntax and semantics of both nodes. The *join* node can have multiple incoming flows and a single outgoing flow, while the *fork* node can have single incoming flows and multiple outgoing flows. The behavioral semantics of both nodes are captured in the specification of δ_{ext} , δ_{int} , δ_{con} , and λ functions. For structural semantics, the model has a list of queues, each corresponding to an input port for holding inputs while waiting for other inputs to arrive through other input ports. Once there is an element in each queue, the model moves into a transitory state to dispatch the output.

The role of the ports in the *SYNC* atomic model captures the syntax of the *join* node. The in_i input ports and the out_i output port correspond to incoming and outgoing flows for the *join* node (see Figure 10.1 (c) and (d)). The behavior specification for the SYNC model (see Listing 10.1) shows the importance of providing structural and behavioral semantics for flows into and out of the UML activity node, and the inclusion of the ports for the SYNC model enhances coupling it to action

nodes. The coupling with ports is more expressive as compared with flows that abstractly connect activity nodes. The same observation applies to the *SELECT* model.

SELECT activity model:

Similar to the *SYNC* model, the *SELECT* model (see Listing 10.2) has one input port for each incoming flow and one output port for each outgoing flow. However, based on the semantics of the *decision* and *merge* activity nodes, the *SELECT* model transitions to an active state as soon as it gets input through one of its input ports. Then, based on specific conditions, it decides which port the output is dispatched from. After determination, the output gets dispatched only through that particular output port. In the case of a merge, the output is always dispatched through the same output port since the model has only one. In other words, the *SELECT* model with multiple input ports and one output port corresponds to a merge node. Moreover, with a single input port and multiple output ports, it corresponds to a decision node. It also corresponds to both if it has multiple inputs and multiple output ports. Figure 10.1 illustrates basic mapping from activities to DEVS.

The *merge* and *decision* nodes are both control nodes, and they are symmetric in terms of their incoming and outgoing flows. The *merge* node receives multiple incoming flows and produces a single outgoing flow, while the *decision* node receives a single incoming flow and produces multiple outgoing flows. From a semantic point of view, they receive or produce flows that their guarding conditions evaluate as true. Only a single flow is selected for a particular I/O. Similar to the case in the *SYNC* model, the behavioral semantics of both the *merge* and *decision* nodes are captured in the *SELECT* model in the specifications of δ_{ext} , δ_{int} , δ_{con} , and λ functions. A list of Boolean values is attached to correspond to the flows for the evaluation. Also, a queue

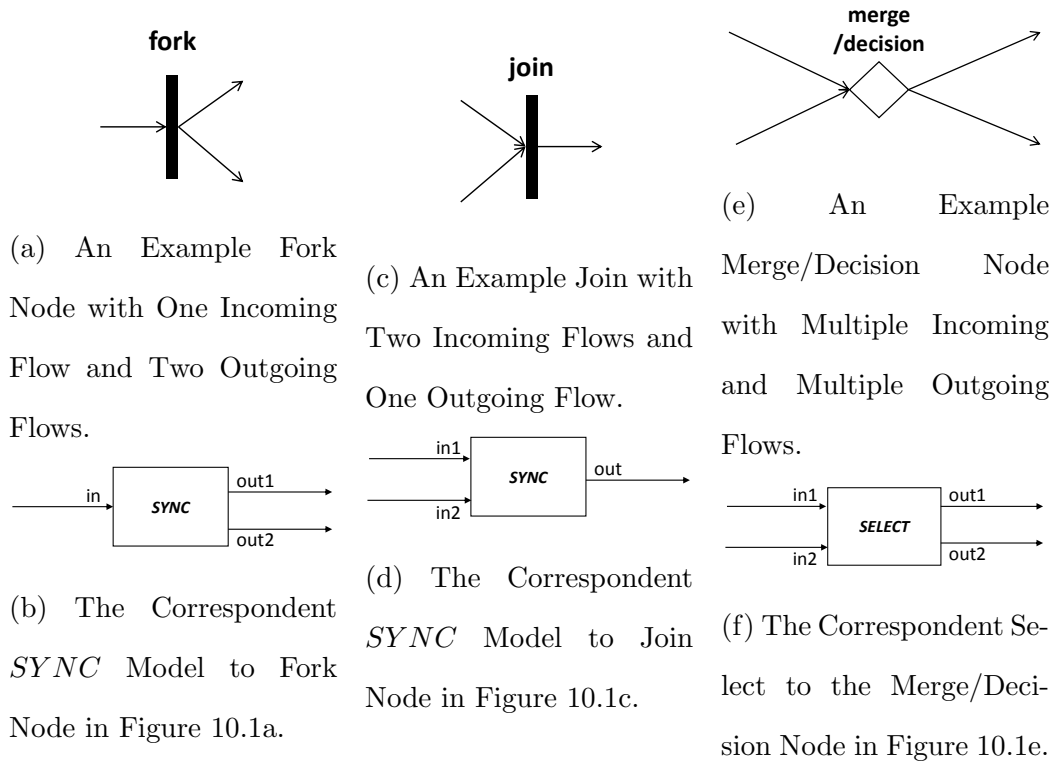


Figure 10.1: Illustration of the Mapping of Different Activity Nodes with Accounts to Multiple Ports and Couplings.

is defined for holding elements in case of receiving inputs while in a busy state. Once an element is received, a transitory state is instantaneously entered before dispatching output.

Coupled activity model:

In the DEVS formalism, each coupling is attached to a port at both ends. For internal coupling, the beginning of the coupling assigns to an output port, and the end assigns to an input port (i.e., the coupling is unidirectional). It is, however, permissible for a port to be attached to multiple couplings. For example, a model A with one output port out can be attached to two couplings c_1 and c_2 where c_1 links the output port out in A to the input port in in model B , and c_2 links the output port out in A to

input port *in* in model *C*. In such a mechanism, any output dispatched from model *A* will be duplicated and simultaneously sent out to both models *B* and *C*. This discipline may appear to be suitable for the SYNC specification or the *fork* node in particular. It allows for a dividing or combining mechanism in model *A*. Conversely, such a mechanism, once needed, ought to also be part of the receiving models that are *B* and *C*. Such a scenario is possible in this example, but it should not be imposed. Therefore, we propose dedicating a single output port for each coupling, as shown in Figure 10.1, to allow for combining or dividing mechanisms to be defined in any one of models *A*, *B*, or *C* or any combinations thereof. Hence, such mechanisms are left undefined in both DEVS and UML. They are both abstract in terms of requiring mechanisms to handle output getting dispatched through outgoing ports or pins (single or multiple) with multiple links attached to them (couplings or flows). In the parallel DEVS formalism, the receipt of multiple inputs through the same input port is possible. However, they operate in an arbitrary order if they arrive at the same time instant. In the UML, defining an input pin with multiple incoming flows imposes join-like semantics dictating that execution should wait for inputs from all incoming flows before proceeding (Eclipse Foundation, 2016b).

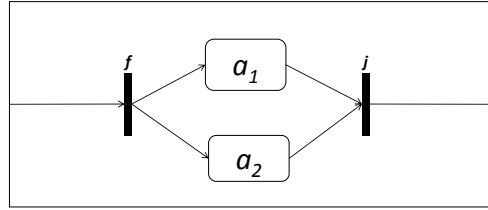
10.3 Exploiting Parallelism

The notion of parallel simulation is exploited in the proposed activity specification based on the parallel, modular, and hierarchical DEVS formalism (Chow, 1996). Parallel discrete event simulation (PDES) is divided into two categories (Fujimoto, 2000). First, conservative approaches are developed based on strictly preventing causality violations. The second category is optimistic PDES, where the simulation allows violation of causality constraints while employing mechanisms to detect them when they happen. The Parallel DEVS simulator (Zeigler *et al.*, 2018b) exploits paral-

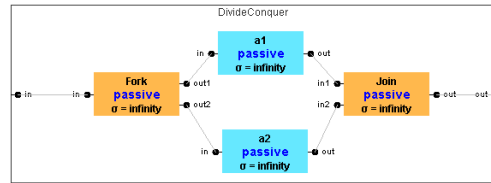
lelism based on the causality constraint as in PDES without carrying out optimistic processing. Events in some models can cause or otherwise influence events in other components. Due to modularity, the components only communicate events through input/output couplings. In each simulation cycle, all imminent components execute in parallel. Once all output events have taken place, all the corresponding influences will execute in parallel for the subsequent processing of input events (received output events). The simulation protocol maintains the variables for the time of last event and time of next event along with output message bags to exchange among components through flat and hierarchical couplings.

10.3.1 Parallelism Semantics

The use of activities stands to serve as a basis for describing different patterns that include parallelism semantics. Encountering such situations is an essential part of developing activities, and therefore, should be examined in a simulation environment. In this section, an example (divide and conquer multiple processor archetype) is selected to demonstrate the use of the first type for control activity nodes (i.e., *fork* and *join* nodes), which demonstrate some aspects of parallelism semantics because they allow for parallel flows to proceed within an activity. They are both shaped with an opaque rectangle (see Figure 10.2). And to a limited degree, they also resemble the semantics of the *transition* concept in Petri nets (Murata, 1989). The *fork* node is used to concurrently split an incoming flow into multiple outgoing flows. The offering of the flow can be arbitrarily accepted by the receiving nodes. The *join* node, on the other hand, receives multiple incoming flows and produces one after synchronizing them. The concurrent flows in a_1 and a_2 are two independent components. However, they synchronize at two junctions, at the flows of events into and out of the *fork* and *join* components.



(a) An Activity Abstraction for the Divide and Conquer Architecture Archetype.



(b) The Initial Simulation View of the Divide and Conquer Architecture.

Figure 10.2: Activity-Based Modeling of the Divide and Conquer Architecture.

Due to the modularity in the DEVS formalism, the representative components for semantics mapping are defined with input and output ports. An atomic model component generally has multiple input and output ports. The simultaneous arrival of a bag of inputs may occur through the same or different ports. In the case of having a coupling between one output port of a model and multiple input ports belonging to another model, the output is duplicated and sent to each of the input ports. Although the formalism does not have a built-in mechanism to prevent the duplication of events, it can be accounted for in the model.

On the one hand, the *fork* node may dedicate an output port for each outgoing flow. On the other hand, one output port can correspond to all outgoing flows. In the former, distinct outputs are sent through different couplings. In the latter, an output is carried through all correspondent couplings, which can be replaced with some other

logic in the model. The expressiveness, complexity, and scale of these approaches can be further examined.

Communicating I/O through coupling is instantaneous. Therefore, inputs arrive at the corresponding models in parallel (i.e., at the same time instance). It is, then, the model's responsibility to process, hold, or lose any input it receives. Multiple inputs can be simultaneously obtained by the same model. The processing can take place if the model is in a "passive" state (i.e., a state in which the model can accept inputs). The holding occurs if the model has some queuing mechanism. The inputs that cannot be processed or otherwise stored are lost.

Another important aspect of parallelism is the handling of event collisions. The input may arrive at the same time instant when output is scheduled to dispatch. The confluence function δ_{con} can handle this collision between input and output. The order was imposed in the classic DEVS formalism using a *select* function where output dispatching precedes the processing of inputs. Moreover, only one atomic model of a coupled model must execute at once; however, in the parallel DEVS formalism (Chow, 1996), this restriction is relaxed. Every atomic model can specify the simultaneous input and output ordering and execution independently of any other atomic model because the constituent atomic models accommodate for the simultaneity of the input and output events. They also account for the unidirectional external input and the internal and external output couplings.

10.3.2 *Simple Experiment for an Archetype Divide and Conquer Architecture in DEVS-Suite Simulator*

An experiment was devised to demonstrate some of the aspects discussed earlier with the DEVS-Suite simulator is used for developing and executing the experiment. The DEVS-Suite simulator is equipped with capabilities such as animations and linear

and superdense time trajectory run-time tracking, and these capabilities are used to observe and monitor key aspects of the behavior of the archetype architecture. First, the divide and conquer architecture is coupled with an experimental frame (EF) model (Rozenblit, 1991). The EF has a simple generator to stimulate the archetype model by sending it inputs. For demonstration purposes, this simple experiment generates outputs every five time units. The transducer is used to analyze the model's properties, such as turnaround time and throughput for processed jobs. The generator communicates with the divide and conquer coupled model (Figure 10.2b) via an external input coupling, and the transmission of a job through each coupling is instantaneous. It is easy to assign a delay to job transmission by, for example, introducing a delay component between the sender and receiver of the job. Alternatively, a delay can be added to the time assigned for processing the job. Once the job arrives at the fork node, the model sets its sigma to a zero time advance. Therefore, it only transitions to a transitory state which instantaneously sends out the job. Essentially, a delay period can be set, as we will discuss further when making observations about timing. Nonetheless, we note that coupling in DEVS is instantaneous. The account for such delay can only be made through the notion of elapsed time or in the time advance function, which is defined within the atomic model.

A dedicated port corresponds to each coupling. However, the output is dispatched simultaneously by the output function in the *fork* component. The components a_1 and a_2 (i.e., processors) receive the inputs simultaneously. If a processor is in the phase *passive*, it transitions to the phase *busy*. To observe a certain behavior, a_1 is set to process inputs five times faster than a_2 . Each processor has a FIFO queue to hold jobs, and the stored jobs are the ones received during phase *busy*. The *join* component is specified to receive both inputs after being processed by the a_1 and a_2 components. When an input arrives on a port, it waits for an input from the

other input port. The received inputs are combined, and then there is the possibility to add delay before dispatching. Note that the state trajectory for the phase of a_2 (Figure 10.3) remains unchanged in phase *busy* because it processes jobs slower than a_1 . The time needed for processing is greater than or equal to the job arrival rate. Therefore, a new job arrives before or immediately after finishing the current one. In this configuration, we set the processing time to be equal to the job arrival rate to illustrate superdense time trajectories, as shown in Figure 10.3 right beneath the phase time trajectory. In such a case, a phase repositioning to *passive* happens at the same time instant of receiving subsequent jobs and therefore it transitions back to *busy*. Note that this repositioning does not appear in the main trajectory because it is instantaneous. The property of a_2 remaining in phase *busy* can be formulated and checked in tools with formal verification capabilities such as UPPAAL. The archetype can also be specified using constrained DEVS (Gholami and Sarjoughian, 2017), and then verified using the DEVS-Suite. It indicates that once a_1 enters the phase *busy*, it remains in this phase forever, which is consistent with the state trajectory shown in Figure 10.3.

10.4 Flow Selection Schemes

In a multiprocessor pipeline architecture, a job may travel through multiple stages before being completed. At each step, the model may have to decide whether the job has been completed or if additional processing is needed. A decision node in Figure 10.4 depicts this choice. Such a node only represents an abstract element for selecting one of its outgoing flows. It has yet to be equipped with certain conditions or perhaps a utility function to facilitate decision-making. In this architecture example, the decision node is concerned about the completion of the job.

Further considerations may take place in this decision logic. Activity diagrams

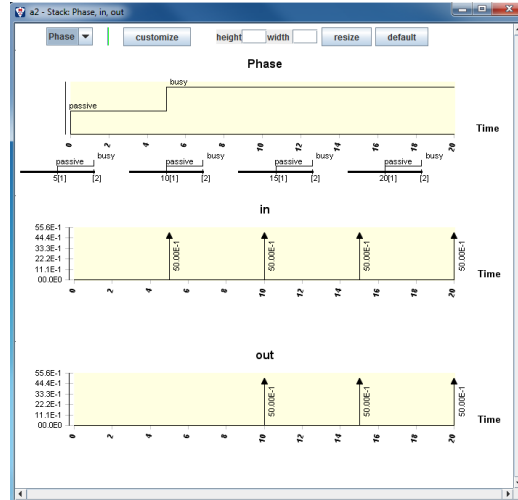


Figure 10.3: The Trajectories for the State Variable $phase$ and the Input in and Output out Ports With Events for the a_2 Component.

alone do not stand to support such elaboration. The outgoing flows from the decision (choice) node are associated with abstract conditions that are left to be elaborated in one or more concrete layers following the MDA concept. In our approach, we examine concretization by developing a simulation of the decision node based on the DEVS formalism. Meanwhile, the separation between the abstract layer/layers and their counterpart concrete ones is iterative and thoroughly maintained.

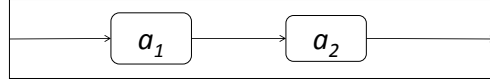
We define the processing stage to be a stage where a job undergoes partial processing accompanied by a delay. Since the job travels through multiple stages, its processing time is simply the total of the delays encountered at each processing stage. Hence, an activity is too abstract when it comes to the concept of time, and neither activity nodes nor edges can account for the delay in the UML metamodel 2.5 (OMG, 2017). In DEVS formalism, the notion of the passage of time is supported for atomic models with dispatching and receiving of events between any two components occurring in order at the same time instant. We propose the use of control nodes to provide a means for the so-called controlled coupling (Alshareef *et al.*, 2018), and we show

that such a control node benefits from a more intuitive, yet rigorous, framework for modeling time-based dynamics of distributed systems.

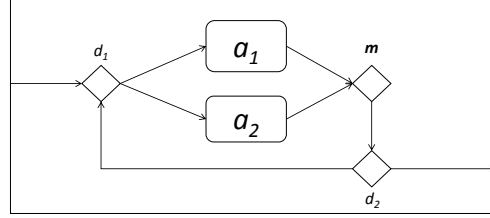
10.4.1 A Pipeline Architecture

As described above, a simple pipeline consists of multiple units for processing a task (a non-trivial job) in a piecemeal fashion and in a particular order. Another way is to introduce decision points among units to determine when the task has been completed and to which unit the task needs to be assigned next. Many aspects of the feedforward and feedback disciplines (Figure 10.4) can be accounted for in both activity and DEVS models.

Considering activity modeling, a key to processing tasks as such is the way nodes are organized to allow the flow of a certain task. Such activity elements can exist in the flow. Each flow can also be characterized by the nodes that precede it and the nodes that follow it. For example, the outgoing flows from a *decision* node can relate to propositions that evaluate to either true or false. This is not necessarily the case with outgoing flows from an *action* or even a *fork* node. Multiple outgoing flows can be produced from the same node (e.g., a *fork* node) as described in the divide and conquer architecture. Hence, in the pipeline with feedback, parallel flows are allowed. In Figure 10.4a, we illustrate the discipline of a pipeline where the task travels through single flows to different elements in a strict sequential order. In Figure 10.4b, we illustrate the decision-making process, where each task also travels through the same elements while allowing parallel flows for multiple actions. In this architecture, a task encounters two decision-making procedures. First, the task is checked before assignment to a certain processor. Second, whether or not the task has been completed is checked. A possible scenario of processing one task starts with the task being checked and assigned to a_1 . After some delay, the task is sent out



(a) A Simple Pipeline.



(b) A Pipeline with Two *Decision* Nodes and a *Merge* Node.

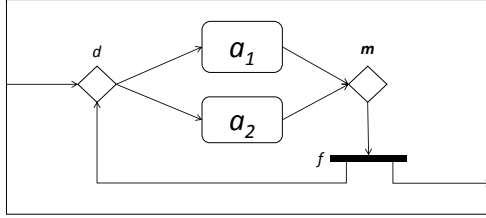
Figure 10.4: Different Abstractions of the Pipeline Architecture with Possibly Different Temporal Attributions in Their Simulations.

from a_1 to the merge node and immediately delivered to decision node d_2 . Decision node d_2 checks whether the task has completed or not. When the task has not completed, it is sent back to d_1 and then assigned for processing at a_2 . Afterward, this processed task directs to the *merge* node and immediately to the decision node d_2 , which dispatches the completed task if it has completed processing the required action nodes. The two pipeline disciplines differ in their activity elements even though they deliver the same outcomes from a the standpoint of timing and simulation. Having the discipline with two decision elements allows for further control over the assignment of tasks to different processors and checking for completion of the tasks. This discipline provides a greater degree of specification of time granularity through the decision and merge nodes (i.e., lifting the restriction on the choices for the incoming and outgoing flows to be instantaneous). Therefore, the decision and merge nodes can be used to devise different pipeline processing procedures. Furthermore, additional measures of performance, such as throughput and buffering, can be computed.

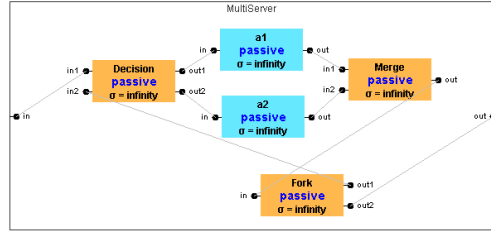
10.4.2 A Multi-Server Architecture

It follows from the previous two architectures, namely the divide and conquer and the pipeline, that all of the essential activity nodes have been discussed and used in both multiprocessing regimes. In this abstraction, the *decision*, *merge*, and *fork* nodes are used (see Figure 10.5). An activity input parameter defines for the activity and is where the *decision* node d receives its incoming flows, and a decision is made to which subsequent node (i.e., a_1 or a_2) the task needs to be sent. Its outgoing flows are subject to satisfying some Boolean conditions. Satisfying the condition redirects the task to the most suitable nodes that can depend on action node availability (see the section 10.5.3 for simulating activities in DEVS-Suite).

Other considerations can be taken into that account to achieve different computational goals. After the completion of the set of actions, the *merge* node m collects the tasks and an output is produced. This acts as a bridge point for the flow toward other nodes and separates the flows from the *fork* node f to avoid synchronization. Because the two processors are independent in this architecture, as opposed to other types, they do not have to be synchronized. Therefore, the activity in Figure 10.5a demonstrates the way a flow can be allowed to proceed without waiting for another. The other alternative is to link outgoing flows from both a_1 and a_2 to f directly, but that would enforce waiting for both flows, whereas that is not necessary for this particular architecture. The *fork* node receives the completed tasks and then produces two outgoing flows. One flow acts as a notification being directed back to the decision node to notify it of the task completion. Thus, the corresponding resource becomes available for processing another task. The other flow goes to the activity output parameter.



(a) An Activity for a Multi-Server Architecture.



(b) The Initial Simulation View of the Multi-Server Model.

Figure 10.5: Activity-Based Modeling of the Multi-Server Architecture.

10.5 Framework for Activity Modeling and Simulation

In an earlier study, Alshareef and Sarjoughian (2017) proposed and developed mapping from the parallel DEVS formalism to the elements of the activity model. The core focus of the mapping was on representing the behavior of the atomic DEVS model. The illustrative metamodel in Figure 10.6 shows important aspects of this relationship, where both the DEVS and activity metamodels are examined. On the one hand, the concept of state change as defined in DEVS (i.e., state transition, output, and time functions) is aligned to the activity node. On the other hand, various notions of the behavioral activity diagram as defined in the UML complement the atomic model. The I/O is also looked at from a DEVS vantage point to set the basis for modular and hierarchical construction of models in the proposed approach.

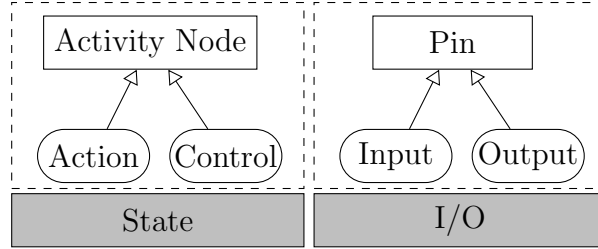


Figure 10.6: A High-Level Sketch Illustrating (A) the Incorporation of Action and Control Node on the One Hand and State on the Other, and (B) a Conceptual Relationship between I/O and Activity Pin.

10.5.1 Time for Activities

The simulation of activities such as those described above can be precise using a well-defined time base (Alshareef and Sarjoughian, 2018b). Explicit temporal specifications such as logical time can eliminate certain ambiguities that manifest themselves during enabling execution. We also described some limitations related to the expressive behavior of the atomic DEVS model, on the one hand, and the precision of the activity model, on the other hand. It is difficult to account for limitations that are rooted in behavioral specification by solely depending on, for example, debugging code techniques and validation methods alone. Such defects particularly arise in behaviors with relatively more complex temporal structures such as the ones that must be characterized using superdense time (Manna and Pnueli, 1992).

When simulating the mentioned archetype above architectures, it is essential to observe certain phenomena, such as ordering and arrival of multiple inputs at any instance of time. Such aspects might result in state transitions that may not be traceable using typical software frameworks and tools. Since architecture complexity is inherent, it is useful to have the means to generate superdense time trajectories for executable models (Sarjoughian and Sundaramoorthi, 2015) and for parallel DEVS

models, in particular. This feature is necessary on multiple occasions, including the model development, testing, and the simulation experiment for different multiprocessor architectures. In all cases, the processing unit or any other component is expected to receive either single or multiple inputs simultaneously. It is also likely to have multiplicity and simultaneity for outputs.

When receiving multiple inputs, their order is arbitrary, and the model may or may not account for that. The external transition function takes place given the current state, in addition to the received bag of inputs and the elapsed time. If there is a state transition due to a single input, then the transition is visible using a linear time trajectory. The situation of zero time advance, as in the divide and conquer archetype example in Figure 10.3, can be tracked using a superdense time trajectory. Such time representation is due to some elements that are added to control the flow without encumbering time delay. Therefore, state transitions that represent such elements can only be made visible using superdense time. The same holds true for similar nodes, especially the ones for control flow purposes only. We will discuss assigning logical execution time to different node types using the time advance function.

It is essential to consider various representations for the notion of time in a simulation environment such as the ones presented by (Goldstein and Khan, 2017). In some cases, limited time-based representations are inadequate when addressing relatively challenging concurrency and synchronization issues. We demonstrate the use of different taxonomies and how they may correspond to activities. The goal is to facilitate earlier experimentation for different processing architectures, with possibly varying lower level manifestations through conforming to MDA guidelines. Identifying a broader set of timing needs can be cumbersome. Thus, it is crucial to facilitate making a more informed decision, especially in cases where efficiency, cost, and scale trade-offs may exist.

10.5.2 Observations of Temporal Analysis with Activities

Notwithstanding the behavioral complexity detailed for multiprocessor archetypes, a temporal analysis may follow, using an activity node classification based on the activity metamodel (OMG, 2017). Different temporal aspects are characterized by which components are used to describe their specifications and different temporal characteristics can ascribe to components based on their specification aspects. For example, some components represent control nodes, such as decision. The time elapsed in these components is defined to be the time spent controlling the flow in some activity. Likewise, the time elapsed in other components is characterized based on the node type for which the specifications of these components ascribe.

Assume each node in the activity models mentioned above is associated with processing time pt , which is either zero or a positive real number. We refer to the activities in Figures 10.2a, 10.4b, and 10.5 as DC, PL, and MP, respectively. We also consider a task that is carried out by one activity holistically and can be assigned to one or divided among multiple activity processing nodes such as those defined for DC. The total time required for the task completion must consume in the processing nodes only. Assume the control nodes may consume time. However, this time cannot count toward task completion. Instead, they account for other time-consuming considerations such as overhead. We formulate such assumptions using the following definitions:

T_{pt} refers to the time required for completing the task or some part thereof.

T_p refers to the time from when processing the task/tasks is initiated until its completion, without accounting for overhead or the time consumed for controlling the flows. Note that when the task is directed to one action only, then T_p is the same as T_{pt} . Also, note that when the task gets assigned to N actions, then T_p will be

equal to T_{pt}/N ad infinitum.

T_c refers to the time consumed by control nodes. Formally, for one component, it is the total time elapsed for the atomic model while in a non-passive phase. Thus, T_c of all control nodes in some activity is the total non-passive time for all atomic models that correspond to these control nodes in that activity. This period may account for dividing tasks into multiple sub-tasks and combining them if necessary. It may also account for synchronization and other timing considerations. Hence, time consumption varies from one architecture to another since the controlling mechanism may differ. In DC, T_c would include the time consumed by *fork* corresponding components and refers to the time required for dividing the task to prepare it for being processed by other parallel components. It also includes the time needed for combining multiple parts (the join) of the task after processing. In PL, T_c would consist of the time consumed by both decision nodes. In the first decision node, it refers to the time required for deciding to which processing node the task needs to be directed. In the second one, it refers to the time needed for determining whether the task has completed or not. In this architecture, T_c would also include the required time for merging flows. In MP, T_c would consist of the time required for deciding to which processing node the task needs to be assigned. It also includes the time needed for merging flows and the time needed for redirecting the completed task and notifying the decision component of the task completion.

$c_{i,active}$ refers to the control node i while being active in managing the flow during the processing of the task.

$a_{i,active}$ refers to the action i while being active in processing the task or a part thereof.

α is the task arrival rate.

The task is processed in either a_1 or a_2 , as in the MP architecture. The task could

also be processed in parallel in a_1 and a_2 , as in the DC architecture, or sequentially, as in the PL architecture. For the above archetypes, time consumption is calculated as $T_{pt}(task) = T_{pt}(a_{1,active}) + T_{pt}(a_{2,active})$. For an archetype with an arbitrary size, the time required for task completion is equal to the total active time of all actions A that carry out the processing, which can be formulated as

$$T_{pt}(task) = \sum_{i=1}^n T_{pt}(a_{i,active}), \quad (10.1)$$

where $n \in \mathbb{N}$ is the number of actions that is, for example, two in the activities mentioned above.

Similarly, T_c is defined to allow accounting for the overhead time in different multiprocessor architectures with different performance schemes. In Figure 10.2a, this accounts for the time consumed by the corresponding atomic models for the *fork* as well as the *join* nodes. For the given DC, the T_c for a given task is defined as $T_c(task) = T_c(c_{1,active}) + T_c(c_{2,active})$. Thus, the time consumed is equal to the total active time of all control nodes, which can be formulated as

$$T_c(task) = \sum_{i=1}^l T_c(c_{i,active}), \quad (10.2)$$

where $l \in \mathbb{N}$ is the number of control nodes in that particular activity.

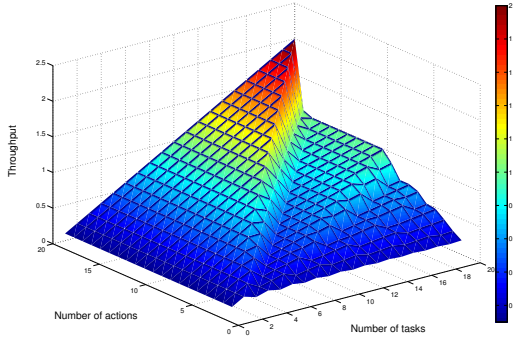
We assume the throughput can be measured based on both T_{pt} and T_c acquired from Eq. 10.1 and 10.2, respectively. It can be used to identify the computational efficiency of each architecture while accounting for the distinction between processing time and other time-consuming elements. This type of difference is accessible through the abstraction of the meta-layer, where control nodes are being defined and then realized concretely in the simulation environment. Such measurements are essential for making critical decisions in various application domains. In formal terms, the

throughput is identified based on the arrival and departure of the tasks to the coupled model that is created to correspond to the processing regime, with activity serving as an abstraction.

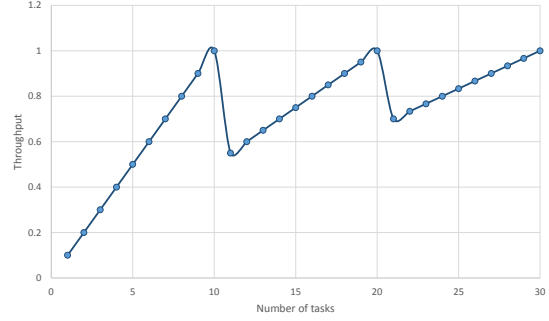
We characterize the time assigned by control nodes to be consumed by task assignment to a particular node, dividing the task into sub-tasks, or synchronizing sub-tasks. We assume these three are all used in the DC architecture since the task has to be divided, assigned, and synchronized. In PL, only two of these mechanisms take place, the division and the task assignment. In MP, only one takes place since the task has to get solely assigned to a particular node in a general multi-processing regime. Thus, a possibility for T_c of each architecture is to be assigned relative to the number of actions.

Particular cases of throughput can be simply observed subject to restrictive assumptions about the configuration of the experiment. For example, the number of tasks and their arrival rates are both significant in determining the throughput, especially with specific exploitation of parallelism. Under strict restrictions, some observations may follow trivially. For instance, the divide and conquer regime will outperform other architectures for one task if T_c consumes zero time in all architectures for each task. Similar observations can take place when different assumptions are given concerning other variables, such as the arrival rate of tasks or T_{pt} .

Other cases of throughput can be calculated under some strict assumptions for T_{pt} and the arrival rate of tasks. In the case of the DC architecture, the throughput can be calculated trivially based on these assumptions by simply dividing the number of actions by the total processing time for the completed tasks. In other words, the throughput is equal to n/T_{pt} . For the other archetypes (i.e., MP and PL), the case is less trivial due to the minimum sequential processing time. Nevertheless, throughput for these is the same as throughput for the DC architecture for the best-case scenario,



(a) Throughput Given Different Number of Actions and Tasks Arriving at the Same Time.



(b) Throughput When the Number of Actions Is Set to Ten ($n = 10$) Given Different Number of Tasks Arriving at the Same Time.

Figure 10.7: Particular Cases of Throughput of the Multiprocessing (MP) and Pipeline (PL) Architecture Are Observed with Different Assignments of the Number of Tasks and Actions. T_{pt} Is Set to 10 Time Units in All Cases and T_c Is Assigned Zero.

where the number of tasks and the number of actions are equal (see Eq. 10.3). The worst case scenario is where there is only one task that leads to less parallelism exploitation and, consequently, resulting in lower throughput. In Figure 10.7, some observations are made where T_p is calculated using the following formula:

$$T_p = \left\lceil \frac{k}{n} \right\rceil T_{pt}, \quad (10.3)$$

where k is the number of tasks, and n is the number of actions. Figure 10.7a shows throughput with different assignments for both n and k . Figure 10.7b shows throughput when $n = 10$ and with different assignments to k . Note that the use of the ceiling is due to the sequential part of the processing. This part has to be at least T_{pt} , resulting in throughput that is always at most one, and the cases of throughput get closer to one as the number of tasks increase, as shown. In the following section, we discuss

other cases where the simulation becomes necessary to arrive at certain results.

10.5.3 Simulating Activities in DEVS-Suite

A set of atomic models corresponds to the discussed DEVS specifications of the activity constructs in the DEVS-Suite simulator. The aim is to create models that complement the previously developed library and tool for creating DEVS models with the activity notation. The library is made as generic and flexible as possible to allow it to account for a broader range of activity-based DEVS models. Currently, the library consists of two generic atomic models by which the primary activity control constructs, in addition to the action, can be realized.

The *decision* node is realized with multiple output ports and an array of Boolean values, where each value corresponds to a single output port. Upon the execution of the output function, some output is dispatched through a designated port when the Boolean condition that corresponds to the output is true. Hence, σ can become assigned with some positive value by δ_{ext} or δ_{int} functions in any atomic model that corresponds to the decision node of any other node. The *merge* node is similarly realized, but with multiple input ports. Note that a node can be instantiated to have both decisions and merge properties as discussed earlier in Listing 10.2 for the *SELECT* model.

The *fork* and *join* nodes are also realized in the *SYNC* atomic model, where the implementation accounts for synchronizing input and outputs. Currently, the combining and dividing processes only account for the timing requirement. It remains an open problem to introduce a combining/dividing mechanism that may suit different semantics. The model corresponding to the *join* node includes multiple queues, where each queue accounts for a certain input port. Storing inputs in such a fashion permits accounting for inputs coming from different models that may arrive through multiple

ports at different time instances. Listings 10.3 and 10.4 demonstrate the external and internal transition functions for the *fork/join* procedure, respectively.

Listing 10.3: The external transition function of *SYNC*

```
parameter: double e, message x
begin
  continue(e)
  for i  $\leftarrow$  1 to x.length
    job  $\leftarrow$  value of  $i_{th}$  message
    j  $\leftarrow$  port number
    add job to  $j_{th}$  queue
  end for
  if all queues are non-empty
    & phase is waiting then
    for i  $\leftarrow$  1 to n
      dequeue from  $i_{th}$  queue
    end for
    holdIn(sending, prep_time)
  end if
  if phase is passive then
    holdIn(waiting,  $\infty$ )
  end if
end
```

Listing 10.4: The internal transition function of *SYNC*

```
begin
  if all queues are empty then
```

```

    passivate
else if all queues are non-empty then
    for i ← 1 to n
        dequeue from  $i_{th}$  queue
    end for
    holdIn(sending, prep_time)
else
    holdIn(waiting,  $\infty$ )
end if
end

```

In the snippets shown in these listings, the atomic model for *join* is equipped with multiple queues to account for the multiple inputs arriving through different ports, along with their order. Note that this presumes the input flows conform to a particular order. When all queues are not empty, the first element of each is supposed to contribute to constituting the output that is to be dispatched. However, it is possible to prioritize elements of a specific queue based on heuristics. It is also possible to do further manipulation of the queue itself and its enqueue/dequeue procedures to satisfy different needs.

We also devised an experiment for the DC architecture to observe throughput under different settings (i.e., the numbers for tasks and actions). In every setting, the experiment initiates by generating all the tasks instantaneously, with ten units of processing time for each task. It is possible to choose different configurations concerning arrival rates for tasks and processing time based on a specific distribution (e.g., uniform distribution). In this particular example, we set T_c relative to the number of actions, assuming more actions require more time to prepare for dividing and then

combining sub-tasks. This setting amounts to the higher throughput encountered with more tasks and fewer actions (see Figure 10.8). The plot shows the best case, with ten tasks and two actions and worse throughput when there are more actions and fewer tasks. The throughput is calculated upon the finishing time, which is precisely the time unit of dispatching the last task by the corresponding coupled model of the DC activity. The final result gets calculated by merely dividing the number of tasks by that total time.

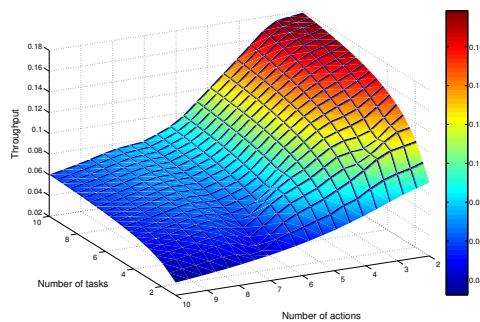


Figure 10.8: Throughput Is Observed by Simulating the Activity of Divide and Conquer in DEVS-Suite, Given Different Numbers of Actions and Tasks Arriving at the Same Time. T_{pt} Is Equal to 10 Time Units in All Cases, and T_c Is Assigned Linearly Relative to the Number of Actions, Where a Greater Number of Actions Requires a Greater T_c Value.

10.6 Conclusion

While remaining at a meta-layer, a variety of elements can be proposed and connected in many different ways. However, these elements might be ineffective or barely meaningful when it comes to concrete realizations or under rigorous transformation conditions. Conversely, lower level implementations cannot benefit from a sophisticated simulation unless an extensive modeling effort takes place. An enormous

amount of cross iterations with expensive endeavors is necessary to confine our choices and hopefully arrive at some useful and concrete simulation for a reasonably small set of the developed higher level constructs.

In this paper, we devised a subset of the activity metamodel and placed it in the context of a discrete event system. We started with a taxonomy of simulation modeling based on MDA and characterized different constructs of activity modeling based on their actual behaviors. We then devised a formal specification for each fundamental element, along with their corresponding implementation in the DEVS-Suite simulator. Different time notions are used to facilitate various temporal analyses. We demonstrated the distinction between control time and processing time. The characterization of activity constructs was used to classify and distinguish between their timing requirements and constraints. We showed the use of activity modeling for different multiprocessing architectures. Our goal is ultimately to be more capable of analyzing computational models by making them subject to experimental designs according to the modeling and simulation principles and guidelines.

REFERENCES

- ACIMS, “CoSMoS”, URL <https://sourceforge.net/projects/cosmosim/>, available at <https://acims.asu.edu/software/cosmos/>, version. 3.0.0. (2017a).
- ACIMS, “DEVS-Suite Simulator”, URL <https://sourceforge.net/projects/devs-suitesim/>, available at <https://acims.asu.edu/software/devs-suite/>, version. 3.0.0. (2017b).
- ACIMS, “DEVS-Suite Simulator”, URL <https://sourceforge.net/projects/devs-suitesim/>, available at <https://acims.asu.edu/software/devs-suite/>, version. 5.0.0. (2019).
- Alshareef, A., “Toward precise semantics of actions”, in “2017 Winter Simulation Conference (WSC)”, pp. 4638–4639 (IEEE, 2017).
- Alshareef, A. and H. S. Sarjoughian, “DEVS specification for modeling and simulation of the UML activities”, in “Proceedings of the Symposium on Model-driven Approaches for Simulation Engineering”, p. 9 (Society for Computer Simulation International, 2017).
- Alshareef, A. and H. S. Sarjoughian, “Model-driven time-accurate devs-based approaches for cps design”, in “Proceedings of the Model-driven Approaches for Simulation Engineering Symposium”, p. 8 (Society for Computer Simulation International, 2018a).
- Alshareef, A. and H. S. Sarjoughian, “Parallelism semantics in modeling activities”, in “Proceedings of the Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium”, (Society for Computer Simulation International (Accepted), 2018b).
- Alshareef, A. and H. S. Sarjoughian, “Metamodeling activities for hierarchical component-based models”, in “Proceedings of the Symposium on Theory and Foundations of Modeling & Simulation Symposium”, (Society for Computer Simulation International, 2019).
- Alshareef, A., H. S. Sarjoughian and B. Zarrin, “An approach for activity-based DEVS model specification”, in “Proceedings of the Symposium on Theory of Modeling & Simulation”, p. 25 (Society for Computer Simulation International, 2016).
- Alshareef, A., H. S. Sarjoughian and B. Zarrin, “Activity-based devs modeling”, *Simulation Modelling Practice and Theory* **82**, 116–131 (2018).
- Alur, R., “Timed automata”, in “Int. Conference on Computer Aided Verification”, pp. 8–22 (Springer, 1999).
- Alur, R., *Principles of cyber-physical systems* (MIT Press, 2015).

- Bartholomew, M. and J. Lee, “Stable models of multi-valued formulas: partial versus total functions”, in “Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning”, pp. 583–586 (AAAI Press, 2014).
- Bedini, F., A. Wichmann, R. Maschotta and A. Zimmermann, “An fUML extension simplifying executable UML models implemented for a C++ execution engine”, in “Proceedings of the Symposium on Model-Driven Approaches for Simulation Engineering”, (Society for Computer Simulation International, 2017).
- Bergero, F. and E. Kofman, “PowerDEVS: a tool for hybrid system modeling and real-time simulation”, *Simulation* **87**, 113–132 (2011).
- Berthomieu, B. and M. Diaz, “Modeling and verification of time dependent systems using time petri nets”, *IEEE transactions on software engineering* **17**, 3, 259–273 (1991).
- Bézivin, J., I. Kurtev *et al.*, “Model-based technology integration with the technical space concept”, in “Metainformatics Symposium”, vol. 20, pp. 44–49 (2005).
- Bocciarelli, P., A. D’Ambrogio, A. Falcone, A. Garro and A. Giglio, “A model-driven approach to enable the simulation of complex systems on distributed architectures”, *Simulation* (2019).
- Borland, S. *et al.*, *Transforming statechart models to DEVS*, Ph.D. thesis, McGill University Libraries (2003).
- Cetinkaya, D., A. Verbraeck and M. D. Seck, “MDD4MS: a model driven development framework for modeling and simulation”, in “Proceedings of the 2011 Summer Computer Simulation Conference”, pp. 113–121 (Society for Modeling & Simulation International, 2011).
- Cetinkaya, D., A. Verbraeck and M. D. Seck, “Model transformation from BPMN to DEVS in the MDD4MS framework”, in “Proceedings of the Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium, Spring Simulation Multi-conference, Orlando, FL, USA”, p. 28 (2012).
- Chidisiuc, C. and G. A. Wainer, “CD++ Builder: an eclipse-based IDE for DEVS modeling”, in “Proceedings of the 2007 spring simulation multiconference-Volume 2”, pp. 235–240 (Society for Computer Simulation International, 2007).
- Cho, S. M. and T. G. Kim, “Real-time DEVS simulation: Concurrent, time-selective execution of combined rt-devs model and interactive environment”, in “Proceeding of 1998 Summer Simulation Conference, Reno, Nevada”, p. 90 (1998).
- Chow, A. C. H., “Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator”, *TRANSACTIONS of the Society for Computer Simulation* **13**, 2, 55–68 (1996).

- Crane, M. L. and J. Dingel, “Towards a formal account of a foundational subset for executable uml models”, in “International Conference on Model Driven Engineering Languages and Systems”, pp. 675–689 (Springer, 2008).
- Damodaran, S. and S. Mittal, “Modeling cyber effects in cyber-physical systems with DEVS”, in “Proceedings of the Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium”, (Society for Computer Simulation International, 2017).
- de Lara, J. and H. Vangheluwe, “Defining visual notations and their manipulation through meta-modelling and graph transformation”, *Journal of Visual Languages & Computing* **15**, 3, 309–330 (2004).
- Derler, P., E. A. Lee and A. S. Vincentelli, “Modeling cyber-physical systems”, *Proceedings of the IEEE* **100**, 1, 13–28 (2012).
- Dill, D. L., “Timing assumptions and verification of finite-state concurrent systems”, in “International Conference on Computer Aided Verification”, pp. 197–212 (Springer, 1989).
- Eclipse Foundation, “Xtext 2.4”, <http://www.eclipse.org/Xtext/documentation/2.4.0/Documentation.pdf>, [Online; accessed 1-July-2015] (2013).
- Eclipse Foundation, “Model Development Tools (MDT)”, URL <http://www.eclipse.org/modeling/mdt/> (2016a).
- Eclipse Foundation, “Papyrus Mars release (1.1.3)”, URL <https://eclipse.org/papyrus/>, available at <https://eclipse.org/papyrus/> (2016b).
- Eclipse Foundation, “Sirius (6.1)”, URL <https://www.eclipse.org/sirius/>, available at <https://www.eclipse.org/sirius/> (2018).
- Fard, M. D. and H. S. Sarjoughian, “Visual and persistence behavior modeling for DEVS in CoSMoS”, in “Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium, Spring Simulation Multi-Conference, Alexandria, VA, USA”, pp. 227–234 (2015).
- Feng, T. H., E. A. Lee and L. W. Shruben, “Ptera: an event-oriented model of computation for heterogeneous systems”, in “Proceedings of the tenth ACM international conference on Embedded software”, pp. 219–228 (ACM, 2010).
- Fondement, F., P.-A. Muller, L. Thiry, B. Wittmann and G. Forestier, “Big metamodels are evil”, in “Model-Driven Engineering Languages and Systems”, pp. 138–153 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013).
- Foures, D., V. Albert, J. C. Pascal and A. Nketsa, “Automation of SysML activity diagram simulation with Model-Driven Engineering approach”, in “Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium”, pp. 1–6 (2012).

- Fujimoto, R. M., *Parallel and distributed simulation systems*, vol. 300 (Wiley New York, 2000).
- Garredu, S., E. Vittori, J.-F. Santucci and B. Poggi, “A survey of model-driven approaches applied to DEVS—a comparative study of metamodels and transformations”, in “Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), 2014 International Conference on”, pp. 179–187 (IEEE, 2014).
- Gelfond, M. and V. Lifschitz, “Representing action and change by logic programs”, *The Journal of Logic Programming* **17**, 2-4, 301–321 (1993).
- Gholami, S. and H. S. Sarjoughian, “Modeling and verification of network-on-chip using constrained-devs”, in “Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS, Virginia Beach, VA, USA”, p. 9 (Society for Computer Simulation International, 2017).
- Giese, H. and S. Burmester, “Real-time statechart semantics”, TechReport tr-ri-03-239, University of Paderborn (2003).
- Goldstein, R. and A. Khan, “A taxonomy of event time representations”, in “Proceedings of the Symposium on Theory of Modeling & Simulation”, p. 6 (Society for Computer Simulation International, 2017).
- Gronback, R. C., *Eclipse modeling project: a Domain-Specific Language (DSL) toolkit* (Pearson Education, 2009).
- Harel, D., “Statecharts: A visual formalism for complex systems”, *Science of computer programming* **8**, 3, 231–274 (1987).
- Harel, D. and E. Gery, “Executable object modeling with statecharts”, in “Proceedings of the 18th international conference on Software engineering”, pp. 246–257 (IEEE Computer Society, 1996).
- Harel, D. and M. Politi, *Modeling reactive systems with statecharts: the STATEMATE approach* (McGraw-Hill, Inc., 1998).
- Henderson-Sellers, B., *On the mathematics of modelling, metamodelling, ontologies and modelling languages* (Springer Science & Business Media, 2012).
- Hollmann, D. A., M. Cristiá and C. Frydman, “CML-DEVS: a specification language for DEVS conceptual models”, *Simulation Modelling Practice and Theory* **57**, 100–117 (2015).
- Hong, J. S., H.-S. Song, T. G. Kim and K. H. Park, “A real-time discrete event system specification formalism for seamless real-time software development”, *Discrete Event Dynamic Systems* **7**, 4, 355–375 (1997).
- Hwang, M. H. and B. P. Zeigler, “Reachability graph of finite and deterministic DEVS networks”, *IEEE Transactions on Automation Science and Engineering* **6**, 3, 468–478 (2009).

- Ighoroje, U. B., O. Maïga and M. K. Traoré, “The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation”, in “Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium”, p. 49 (Society for Computer Simulation International, 2012).
- Jensen, K. and G. Rozenberg, *High-level Petri nets: theory and application* (Springer Science & Business Media, 2012).
- Kapos, G.-D., V. Dalakas, M. Nikolaidou and D. Anagnostopoulos, “An integrated framework for automated simulation of sysml models using devs”, *Simulation* **90**, 6, 717–744 (2014).
- Kim, S., H. S. Sarjoughian and V. Elamvazhuthi, “DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring”, in “Proceedings of the 2009 Spring Simulation Multiconference”, SpringSim ’09, pp. 161:1—161:7 (Society for Computer Simulation International, 2009).
- Kirshin, A., D. Dotan and A. Hartman, “A UML simulator based on a generic model execution engine”, in “MoDELS Workshops”, vol. 4364, pp. 324–326 (2006).
- Lamport, L., “The mutual exclusion problem: Part ia theory of interprocess communication”, *Journal of the ACM (JACM)* **33**, 2, 313–326 (1986).
- Lamport, L., “A simple approach to specifying concurrent systems”, *Communications of the ACM* **32**, 1, 32–45 (1989).
- Latella, D., I. Majzik and M. Massink, “Towards a formal operational semantics of UML statechart diagrams”, in “Formal Methods for Open Object-Based Distributed Systems”, pp. 331–347 (Springer, 1999).
- Lee, J. and R. Palla, “System f2lp—computing answer sets of first-order formulas”, in “International Conference on Logic Programming and Nonmonotonic Reasoning”, pp. 515–521 (Springer, 2009).
- Lei, Y., W. Wang, Q. Li and Y. Zhu, “A transformation model from DEVS to SMP2 based on MDA”, *Simulation Modelling Practice and Theory* **17**, 10, 1690–1709, URL <https://doi.org/10.1016/j.simpat.2009.08.003> (2009).
- Lifschitz, V., “Answer set planning”, in “International Conference on Logic Programming and Nonmonotonic Reasoning”, pp. 373–374 (Springer, 1999).
- Lynch, N., R. Segala and F. Vaandrager, “Hybrid i/o automata”, *Information and computation* **185**, 1, 105–157 (2003).
- Manna, Z. and A. Pnueli, “The temporal logic of reactive and concurrent systems”, Springer (1992).
- Mayerhofer, T., “Testing and debugging UML models based on fuml”, in “2012 34th International Conference on Software Engineering (ICSE)”, pp. 1579–1582 (IEEE, 2012).

- Mayerhofer, T., P. Langer, M. Wimmer and G. Kappel, “xmof: Executable dsmls based on fuml”, in “International Conference on Software Language Engineering”, pp. 56–75 (Springer, 2013).
- McNeill, K., “Metamodeling with EMF: Generating concrete, reusable java snippets”, <http://www.ibm.com/developerworks/library/os-eclipse-emfmetamodel/>, [Online; accessed 1-July-2015] (2008).
- Mellor, S. J., M. Balcer and I. Foreword By-Jacoboson, *Executable UML: A foundation for model-driven architectures* (Addison-Wesley Longman Publishing Co., Inc., 2002).
- Miller, J. and J. Mukerji, “MDA guide version 1.0. 1, Object Management Group”, <http://www.omg.org/docs/omg/03-06-01> (2003).
- Misra, J., *A Discipline of Multiprogramming* (Springer, 2001).
- Mittal, S. and J. L. R. Martín, “Model-driven systems engineering for netcentric system of systems with DEVS unified process”, in “Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World”, pp. 1140–1151 (IEEE Press, 2013a).
- Mittal, S. and J. L. R. Martín, *Netcentric system of systems engineering with DEVS unified process* (CRC Press, 2013b).
- Mittal, S., J. L. Risco-Martín and B. P. Zeigler, “DEVSMML: automating devs execution over SOA towards transparent simulators”, in “Proceedings of the 2007 spring simulation multiconference-Volume 2”, pp. 287–295 (Society for Computer Simulation International, 2007).
- Moallemi, M. and G. Wainer, “Designing an interface for real-time and embedded devs”, in “Proceedings of the 2010 Spring Simulation Multiconference”, p. 137 (Society for Computer Simulation International, 2010).
- Mooney, J. and H. S. Sarjoughian, “A framework for executable UML models”, in “Proceedings of the 2009 Spring Simulation Multiconference”, pp. 1–8 (Society for Computer Simulation International, 2009).
- Mosterman, P. J. and J. Zander, “Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems”, *Software & Systems Modeling* **15**, 1, 5–16 (2016).
- Muram, F. U., H. Tran and U. Zdun, “Automated mapping of UML activity diagrams to formal specifications for supporting containment checking”, in “Proceedings 11th Inter. Workshop on Formal Eng. approaches to SW Components and Architectures, FESCA 2014, Grenoble, France.”, pp. 93–107 (2014).
- Murata, T., “Petri nets: Properties, analysis and applications”, *Proceedings of the IEEE* **77**, 4, 541–580 (1989).

- Nikolaidou, M., V. Dalakas, L. Mitsi, G. D. Kapos and D. Anagnostopoulos, “A SysML profile for classical DEVS simulators”, in “Proceedings of the 2008 The Third International Conference on Software Engineering Advances”, pp. 445–450 (2008).
- Nikolaidou, M., G.-D. Kapos, A. Tsadimas, V. Dalakas and D. Anagnostopoulos, “Challenges in SysML model simulation”, *Advances in Computer Science: an Intl. Journal* **5**, 4, 49–56 (2016).
- NIST, “Framework for cyber-physical systems”, <https://www.nist.gov/el/cyber-physical-systems> (2016).
- OMG, “Unified Modeling Language version 2.0”, URL <https://www.omg.org/spec/UML/2.0> (2005).
- OMG, “Unified Modeling Language version 2.5”, URL <https://www.omg.org/spec/UML/2.5> (2012).
- OMG, “Semantics of a Foundational Subset for Executable UML Models (fUML) version 1.1”, URL <https://www.omg.org/spec/FUML/1.4> (2013).
- OMG, “Unified Modeling Language version 2.5.1”, URL <https://www.omg.org/spec/UML/2.5.1/> (2017).
- OMG, “Semantics of a Foundational Subset for Executable UML Models (fUML) version 1.4”, URL <https://www.omg.org/spec/FUML/1.1> (2018).
- Ozmen, O. and J. Nutaro, “Activity diagrams for DEVS models: a case study modeling health care behavior (WIP)”, in “Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium”, pp. 1006–1011 (2015).
- Pasqua, R., D. Foures, V. Albert and A. Nketsa, “From sequence diagrams UML 2.x to FD-DEVS by model transformation”, in “European Simulation and Modelling Conference”, pp. 37–43 (2012).
- Rafe, V. and A. Rahmani, “Formal analysis of workflows using uml 2.0 activities and graph transformation systems”, *Theoretical Aspects of Computing-ICTAC 2008* pp. 305–318 (2008).
- Risco-Martín, J. L., M. Jesús, S. Mittal and B. P. Zeigler, “eUDEVS: Executable UML with DEVS theory of modeling and simulation”, *Simulation* **85**, 11-12, 750–777 (2009).
- Risco-Martín, J. L., S. Mittal, J. C. Fabero, P. Malagón and J. L. Ayala, “Real-time hardware/software co-design using DEVS-based transparent M&S framework”, in “Proceedings of the Summer Computer Simulation Conference”, p. 45 (Society for Computer Simulation International, 2016).

- Rozenblit, J. W., “Experimental frame specification methodology for hierarchical simulation modeling”, *International Journal Of General System* **19**, 3, 317–336 (1991).
- Sarjoughian, H., “Restraining complexity and scale traits for component-based simulation models”, in “Proceedings of the 2017 Winter Simulation Conference”, (IEEE Press, 2017).
- Sarjoughian, H. S., A. Alshareef and Y. Lei, “Behavioral DEVS metamodeling”, in “Proceedings of the 2015 Winter Simulation Conference”, pp. 2788–2799 (IEEE Press, 2015).
- Sarjoughian, H. S. and V. Elamvazhuthi, “CoSMoS: a visual environment for component-based modeling, experimental design, and simulation”, in “Proceedings of the 2nd international conference on simulation tools and techniques”, p. 59 (ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009).
- Sarjoughian, H. S. and S. Gholami, “Action-level real-time DEVS modeling and simulation”, *Simulation* **91**, 10, 869–887 (2015).
- Sarjoughian, H. S., S. Gholami and T. Jackson, “Interacting real-time simulation models and reactive computational-physical systems”, in “Proceedings of the 2013 Winter Simulation Conference”, pp. 1120–1131 (IEEE Press, 2013).
- Sarjoughian, H. S. and A. M. Markid, “EMF-DEVS modeling”, in “Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium, Spring Simulation Multi-Conference, Orlando, FL, USA”, p. 19 (Society for Computer Simulation International, 2012).
- Sarjoughian, H. S. and S. Sundaramoorthi, “Superdense time trajectories for DEVS simulation models”, in “Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium”, pp. 249–256 (Society for Computer Simulation International, 2015).
- Sarjoughian, H. S. and B. Zeigler, “DEVSTJAVA: Basis for a DEVS-based collaborative M&S environment”, *Simulation Series* **30**, 29–36 (1998).
- Schulz, S., T. C. Ewing and J. W. Rozenblit, “Discrete event system specification (DEVS) and StateMate StateCharts equivalence for embedded systems modeling”, in “Proceedings Seventh IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2000. (ECBS 2000)”, pp. 308–316 (2000).
- Seo, C., B. P. Zeigler, R. Coop and D. Kim, “DEVS modeling and simulation methodology with MS4 Me software tool”, in “Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium”, p. 33 (Society for Computer Simulation International, 2013).
- Shoham, Y., “Time for action”, in “Proceedings of IJCAI”, vol. 89 (1989).

- Simon, H. A., “The architecture of complexity”, *Proceedings of the American Philosophical Society* **106**, 6, 467–482 (1962).
- Soley, R. and the OMG Staff Strategy Group, “Model driven architecture”, OMG white paper **308** (2000).
- Stahl, T., M. Voelter and K. Czarnecki, *Model-driven software development: technology, engineering, management* (John Wiley & Sons, 2006).
- Steinberg, D., F. Budinsky, E. Merks and M. Paternostro, *EMF: Eclipse Modeling Framework* (Pearson Education, 2008).
- Störrle, H. and J. Hausmann, “Semantics of UML 2.0 activities”, in “Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing”, (2004).
- Störrle, H., J. H. Hausmann and U. Paderborn, “Towards a formal semantics of uml 2.0 activities”, in “In Proceedings German Software Engineering Conference, volume P-64 of LNI”, pp. 117–128 (2005).
- Sztipanovits, J., X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras and S. Wang, “Toward a science of cyber–physical system integration”, *Proceedings of the IEEE* **100**, 1, 29–44 (2011).
- Wang, Q. and F. E. Cellier, “Time windows: An approach to automated abstraction of continuous-time models into discrete-event models”, in “AI, Simulation and Planning in High Autonomy Systems, 1990., Proceedings.”, pp. 204–211 (IEEE, 1990).
- Wilkinson, B. and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, vol. 2 (Prentice hall New York, 2005).
- Wymore, A. W., *Model-based systems engineering*, vol. 3 (CRC press, 1993).
- Yonglin, L., W. Weiping, L. Qun and Z. Yifan, “A transformation model from DEVS to SMP2 based on MDA”, *Simulation Modelling Practice and Theory* **17**, 1690–1709 (2009).
- Zeigler, B. and H. S. Sarjoughian, *Guide to modeling and simulation of systems of systems* (Springer Science & Business Media, 2012).
- Zeigler, B. P., J. W. Marvin and J. J. Cadigan, “Systems engineering and simulation: converging toward noble causes”, in “Proceedings of the 2018 Winter Simulation Conference”, pp. 3742–3752 (IEEE Press, 2018a).
- Zeigler, B. P., A. Muzy and E. Kofman, *Theory of modeling and simulation: discrete event and iterative system computational foundations* (Academic press, 2018b), third edn.

- Zeigler, B. P., H. Praehofer and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems* (Academic press, 2000).
- Zeigler, B. P., H. Sarjoughian and W. Au, “Object-oriented DEVS”, in “11th SPIE”, pp. 100–111 (1997).
- Zeigler, B. P. and H. S. Sarjoughian, “Introduction to DEVS modeling and simulation with java: Developing component-based simulation models”, Arizona State University (2003).
- Zeigler, B. P. and H. S. Sarjoughian, *Guide to Modeling and Simulation of Systems of Systems* (Springer, 2017), second edn.
- Zhu, Z., Y. Lei, A. Alshareef, H. Sarjoughian and Y. Zhu, “Domain specific meta-modeling for deep semantic composability”, *IEEE Access* **6**, 18276–18289 (2018).
- Zhu, Z., Y. Lei, Y. Zhu, A. Alshareef and H. S. Sarjoughian, “A unifying framework for UML profile-based cognitive modeling: Development and experience”, in “Proceedings of the 10th EAI International Conference on Simulation Tools and Techniques”, pp. 1–10 (ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017).
- Zinoviev, D., “Mapping DEVS models onto UML models”, in “DEVS Symposium, Spring Simulation Multiconference”, (2005).

APPENDIX A
OTHER CONTRIBUTIONS

Section A.1 includes a demo which I presented in Spring Simulation Multi-Conference 2017 demo session. Section A.2 is a Ph.D. colloquium, which I presented in the Winter Simulation Conference 2017.

A.1 Infusing Simulatability into Software Models

Abstract

The emergence of autonomous vehicles, also known as self-driving vehicles, poses new research challenges to ensure system-wide safety property. In such time-critical Cyber-Physical Systems, verification of software plays a crucial role while interacting with physical parts. We demonstrate a Model-Driven approach using UML activities for modeling a hypothetical scenario for a vehicle system. UML activity models implemented as DEVS simulation models can help reveal certain time-sensitive safety properties.

A.1.1 Transforming Activity Models to DEVS Models: Autonomous Vehicles

Recent advances in UML activity modeling cast them to the DEVS simulation model Alshareef and Sarjoughian (2017). Such work highlights the key role that accurate representation and manipulation of time have for more precise execution of software behavior. We have created a generic library for simulating UML activities utilizing the current architecture of parallel DEVS-Suite simulator ACIMS (2017b). By doing so, the modeler can benefit from the DEVS as a modeling formalism and yet be able to obtain time-accurate execution manifested by the underlying simulator. The semantics of the activities are captured in a set of generic atomic models. These models can be therefore specialized exploiting polymorphism to have specific instances that collectively encompass activity models.

The library currently consists of modularized packages where each has a generic and polymorphic set of atomic models (see Table A.1). Packages also contain some complementary features to enable the use of the set of atomic models such as interpretation and instantiation. The models are formalized first given the DEVS formalism. Then, each atomic model is devised in a way to resemble its corresponding counterpart in the activity metamodel. For example, the semantics of the decision node are formulated in the decision node atomic model generically and then specialized for the specific instance thereof.

Table A.1: A Set of Atomic and Coupled Models for Activities DEVS Modeling and Simulation.

| | | |
|---------------|-----------------|------------------|
| Activity Node | Executable Node | Control Node |
| Object Node | Action | Expansion Region |
| Initial Node | Final Node | Fork Node |
| Decision Node | Merge Node | Join Node |

All models have been realized in a way to remain consistent and benefit from our existing activities metamodel. The metamodel has been built based on the Eclipse

Modeling Framework (EMF) and Ecore. In Figure A.1, we show how the implementation of the library is currently integrated within the architecture of the DEVS-Suite simulator.

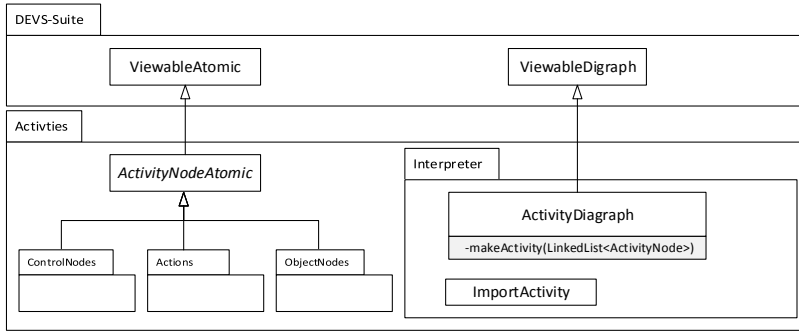


Figure A.1: The Integration of the New Packages Within the Current Architecture.

Figure A.2 summarizes the details about our proposition in practice. The activity model represents an intersection to direct the approaching vehicles and reports collisions that occur in the intersection. The safety of vehicles entering and exiting the crossing can be verified through simulating time-accurate activities of the software dynamically detecting and controlling sensory and actuating signals.

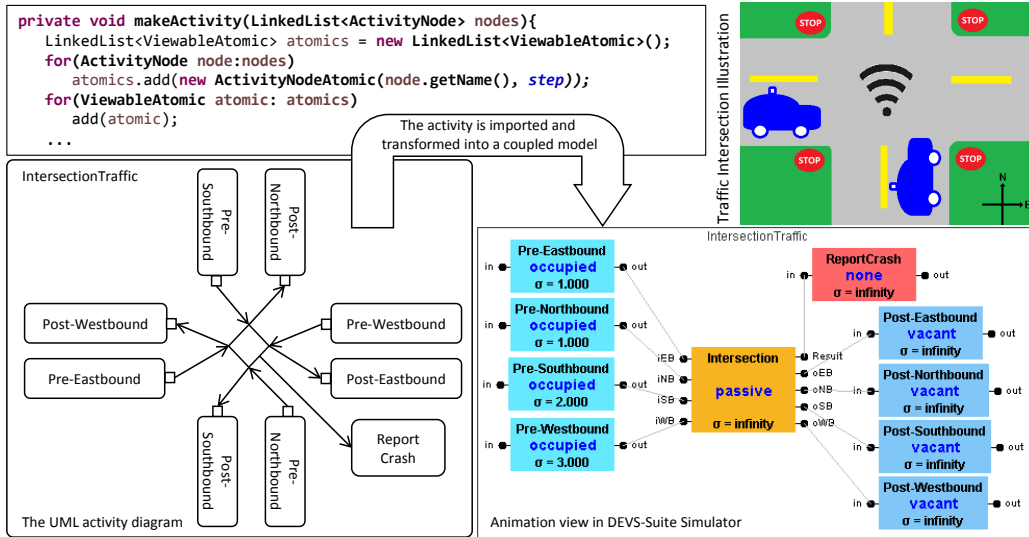


Figure A.2: The Multiple Views for Modeling and Simulation of an Intersection.

A.2 Toward Precise Semantics of Actions

Abstract

Action is the fundamental unit of behavioral specification in models. We propose the use of Discrete Event System Specification (the DEVS formalism) to specify the

semantics of actions. Then, the coupling is used to form different kinds of behavioral models. The statecharts and activities are two different approaches by which the system behavior can be described. Actions are at the core of these two approaches. Their specifications can collectively serve as a significant part of the overall behavior alongside with behavior of other elements such as control. Thus, we propose an approach introducing the concepts of time and state as defined in DEVS for actions; these serve as an abstraction for modeling a wide range of systems.

A.2.1 Introduction

It is essential to use abstractions, languages, and metamodels for behavioral specifications to overcome complexity and scale demand. Any tiny change in the behavioral specification of a model may result in vastly different dynamics. Thus, it is necessary to tame this intrinsic characteristic in behavioral modeling. The problem is approached through how the behavioral specifications are described. Using an ad-hoc approach may not scale, mainly due to the complexity arising from a mesh of actions.

A.2.2 The Atomic Model and the Action

The goal is ultimately to create a means for actions to be specified as precise and flexible as possible for systems that interact in arbitrary, but well-formed, fashion. When modularity is maintained, the system can grow according to the principles of coupling and composability. Thus, we propose DEVS specification creating a correspondent atomic model for each action (see Figure 1). The formal specification of the atomic model $DEVS = (X_M, Y_M, S, ext, int, con, ta)$ is defined with respect to the semantics of the corresponding action. In conjunction with the other defined atomic models for describing a certain semantics for some behavioral diagram element, such as decision node Alshareef and Sarjoughian (2017), these elements can together formulate the correspondent chains of actions (i.e., coupled models) Sarjoughian (2017). The processing time represents a broader range of execution semantics, including an execution step. The time advance function is utilized in the correspondent action model to establish the linkage to the time base. It is either continuous, discrete, or some other as discussed in the literature in some DEVS variant formalisms. Such restrictions can be leveraged to work around some computational compromises to satisfy specific needs.

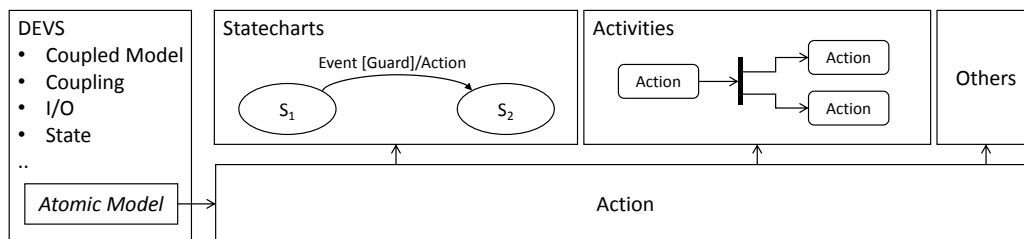


Figure A.3: The Action Abstraction Is Situated at the Heart of Many Behavioral Specifications and Thus Used As a Bridge Between the Formal Specification and Other Semi-Formal or Informal Modeling Approaches.

A.2.3 A Processor Model

We discuss an Activity-based DEVS abstraction specified for a processor model described in Zeigler *et al.* (2000). The processor receives a bag of inputs and distinguishes between them for different kinds of communication. Data is commonly exchanged as well as control according to their intended purposes. Instances of some classes with variables are created at some locus. Therefore, inputs can be used for communicating information about the model to different components. Although using the notion of time may significantly differ in the context of action, the processing time can take various values. It then can model ordering or concurrency semantics such as in the join node defined by the activity abstraction. It also may refer to duration assuming that actions are not instantaneous. Another aspect is to consider the instant of time in which an action may occur or start. Since the action is deemed to be the fundamental unit of behavioral specification, their influence on the state is specified explicitly to be able to provide guarantees across the system of interest. This elaboration is certainly beneficial, especially when considering the possibility to extend the notion of action with time. This forms a strong basis toward achieving the goal of having a precise time-based semantics for actions.

The processor remains idle while not receiving inputs. It may store received inputs in unitary storage or multiple inputs in a queue. Actions also have access to other resources. Their situation in the context of the object model is yet to be examined. In the UML OMG (2012), the action can be created within a context of behavior classifier. Along with control elements, they constitute the overall behavior. Different mechanisms of control then reveal the nature of the behavioral model, as shown in Figure 1. As far as the action is concerned, it can initially or subsequently takes some inputs. Then, the semantics of the action takes place considering these inputs, causing some changes on the state and possibly producing outputs.

To conclude, the capability of developing the behavioral specification in stages is beneficial but yet quite challenging. In our approach, we investigate metamodels, frameworks, and tools to increase accessibility without compromising our models or any of the artifacts thereof. The precise semantics of actions can take place as an essential step in enabling simulation-based studies. Along with supporting control structures, useful simulations for Systems of Systems can be attained.