Modular Simulator for DAE-Based Systems Using DEVS Formalism

Aya Attia[a], Clément Foucher[b] and Luiz Fernando Lavado Villa[c]
LAAS-CNRS, Université de Toulouse, UPS, Toulouse, France
fl

Keywords: Theory of Modeling and Simulation, Differential-Algebraic Equation, Model Transformation, DEVS, Symbolic Computations, Graph Grammar, Load Flow Analysis.

Abstract: Modeling and Simulation of dynamic structure systems present some difficulties, particularly those represented by Differential-Algebraic Equations (DAEs), as structural changes often require modifying equations. To address this, we propose a methodology to build simulators based on Theory of Modeling and Simulation, laying as a foundation for rigorously handling such systems. Our proposal consists of a modular, domain-independent simulator where system's behavior can be represented by DAEs. Systems are represented as graphs, dynamically updated at each step based on predefined scenarios. The simulator automatically generates and processes equations using transformation rules applied to input graphs. To demonstrate the feasibility of our proposal, we apply it to the domain of power systems, particularly the Load Flow Analysis process.

1 INTRODUCTION

Dynamic structure systems are characterized by a composition and internal connections that can evolve over time. The ever-changing nature of dynamic structure systems leads to difficulties to provide a formal representation that can be adjusted to fit system's structural changes, notably, in systems where models are represented by DAEs. Adapting to these structural changes often requires modifying the equations themselves, which represent a break in the simulation flow. Power systems are a concrete example of dynamic structure systems that can be represented by DAEs. These systems may undergo structural changes when there is a fault, a black start or a grid expansion. This challenge highlights the need for formal approaches that provide seamless integration of mathematical rigor with practical adaptability.

The Theory of Modeling and Simulation (Zeigler et al., 2018) provides a robust framework for this purpose. Notably, the Discrete Event System Specification (DEVS) formalism provides a root approach to build models. Despite the theoretical strength of the DEVS formalism, in practice its implementation remains a significant challenge. Due to the very low level of the formalism, it may be hard to provide a mathematical description of the model, therefore, developers usually go straight to the implementation without building a formal model (Blas et al., 2023). This difficulty highlights a recurring issue: the gap between a strong mathematical formalism and its actual concrete implementation.

To address this, our goal is to take a formal approach for the conception of a generic simulator adapted to various fields of applications. This simulator should be conceived in a way that allows it to handle structure changes in the systems using a set of rules. In this article, we present the first step toward this approach: the root structure of a simulator built using the DEVS formal specification. It bridges the gap between theoretical formalism and practical implementation, offering a structured method for creating models and implementing them. To achieve this, we base our simulator on the DEVS formalism by providing a generic library of DEVS components that can be assembled to create the simulator. Input models of the simulator are represented as graphs, a powerful yet simple formalism for representing components and their connections. Finally, using model transformations directly within the simulator, the graphs are turned into DAEs and processed using symbolic computations. Symbolic computations ensure that executable description of system's components is conform to formal one. Our proposal can be tailored to various domains using the provided components that can be customized to carry on the specific computations needed by a particular domain.

[a] https://orcid.org/0000-0002-8610-1851
[b] https://orcid.org/0000-0001-9566-4173
[c] https://orcid.org/0000-0002-7157-5106

Attia, A., Foucher, C. and Villa, L. F. L.
Modular Simulator for DAE-Based Systems Using DEVS Formalism.
DOI: 10.5220/0013514100003970
In *Proceedings of the 15th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2025)*, pages 270-277
ISBN: 978-989-758-759-7; ISSN: 2184-2841

2 BACKGROUND AND RELATED WORK

This section provides an overview of the key concepts and techniques underlying our proposed approach.

2.1 The Theory of Modeling and Simulation

Complex systems modeling and simulation represent a major challenge in various fields. The Theory of Modeling and Simulation (TM&S) provides a structured approach using formalisms such as DEVS. DEVS is a modular formalism that uses discrete events to trigger computations at specific points in time. However, as it is very low-level, it may be hard to express complex models purely in terms of DEVS models. DEVS has been provided various extensions, and we will here use the Parallel-DEVS (PDEVS) variant (Chow and Zeigler, 1994) that allows for parallelism in models execution.

In the TM&S approach, the term *model* can refer to two representations of a system: **formal model** or **executable model**. The **formal model**, also known as mathematical or abstract model, is a type of representation that follows the mathematical grammar of the formalism in order to provide a formal representation of the system model. On the other hand, the **executable model**, or concrete model, is derived by translating abstract models into computer programs, which are then integrated with a simulator capable of executing the defined functions of these models. The formal definition of a PDEVS model uses the following description: $M =< X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >$

In a DEVS formalism, a model is defined by sets and functions. The sets include the inputs X, the outputs Y, and the internal state S. The behavior of the model is described through core functions that govern its dynamics. The external transition function (δ_{ext}) updates the system's state in reaction to input events. The internal transition function (δ_{int}) manages the system's autonomous transitions and is invoked when the time allocated to a state ($t_a(s)$) expires, also triggering the output function (λ). As the PDEVS formalism is an extension of DEVS that handles concurrent events, it uses the confluent transition function (δ_{con}) to resolve conflicts.

Formal models are translated to executable code that can be run using simulators. The goal of simulators is to reproduce the behavior of a real system in a virtual environment, using models, input data, and specific parameters. TM&S constituents and their relationships can be represented by Figure 1. In this figure, we explicitly make the difference between the formal model, which uses the mathematical syntax of the meta-model, and the executable model which is an implementation of the abstract model within a software simulation tool. However, it is often the case that the formal model definition is skipped in favor of describing the model directly into the simulator (Blas et al., 2023). While this should not be an issue if the simulator conforms to the formalism, problems arise when the simulator allows for some flexibilities that are not present in the formalism. The modeler can then be at risk of accidentally breaking the rules of the formalism without noticing it, which can lead to fundamentally incorrect models that behave in an unpredictable way over the simulation course.

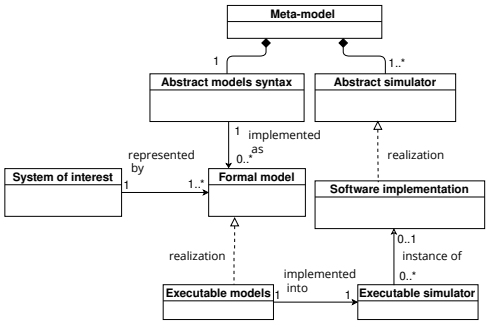

Figure 1: TM&S structural diagram.

2.2 Symbolic Computations

Symbolic computation, or computer algebra, deals with the automatic generation using computer algorithms of mathematical equations, functions and formulas based on symbols and algebra rules (Buchberger, 1996). Through the use of symbolic computation, mathematical expressions can be represented with the same strictness as they would be in an abstract model. Symbolic computations can be provided by various software tools like SymPy (Meurer et al., 2017), Mathematica (Trott, 2007), Maple (Caravantes et al., 2019), etc. In this work, we chose to use symbolic equations as the base data that will be acted on by our simulator. The components of the simulator will communicate data between each other in the form of equations, variables and values that they will use to compute the results of the simulation.

2.3 Graphs

Graphs (Bickle, 2020) are mathematical structures that can be used to model complex systems in various domains. The modeling process using graphs consists in sets of vertices, also called nodes, which represent system components, and edges, also called links, which represent the relationships between system's

components. The interactions between components are provided by the incidence function ψ which maps an edge to a pair of vertices. Graph grammar (Xu, 2013) is described by: $G = (V, E, \psi)$, where V and E are finite sets that represent the vertices and edges respectively, and ψ a mapping function with domain E and co-domain $V \times V$.

Combining graph grammar and Chomsky's generative grammars (Chomsky, 1956), it is possible to introduce dynamic structure to the system thanks to production rules that define how the system's structure and composition change over time (Bouassida Rodriguez et al., 2010). These evolving graphs can be represented as follows: $G =< AX, NT, T, P >$, where AX is the axiom, NT denotes the set of non-terminal vertices, T the set of terminal vertices, and P the set of transformation rules. Figure 2 visualizes the concept of graph grammar and its transformation using Chomsky's generative grammars.

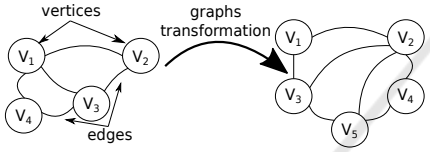

Figure 2: Graph grammar and its transformation.

We chose graphs as the base grammar for systems representation for their simplicity and ability to represent system changes based on transformation rules.

2.4 Power Systems and LFA

Power systems is a field of electrical engineering that studies energy distribution and management (Weedy et al., 2012), typically involve three main elements: **electrical components** (such as generators and loads) that either consume or produce electrical energy, **buses** that interconnect nearby electrical components, and **transmission lines** that ensure the transfer of electrical flow between buses over long distances.

Load Flow Analysis (Albadi and Volkov, 2020), also known as power flow analysis, is a method used in the electrical engineering field to evaluate and analyze the behavior of electrical networks. This technique aims to check the steady-state of operating conditions of a power system. A power system is characterized by the following variables: **voltage magnitudes** V (the amplitude of voltage at a bus), **angles** δ (the phase shift of a bus voltage relative to a reference bus), **active power** P (the power transmitted to system loads), and **reactive power** Q (the non-useful power required to maintain system operation).

The generated results of Load Flow Analysis are used to make important decisions in planning, designing and operating electrical networks. In particular, equipment sizes, load flows optimization and evaluation of the stability of the system in response to load variations and external perturbations.

Many simulators were developed in order to model power systems, including many based on MATLAB/Simulink such as (Chaturvedi, 2017). Another example of power systems modeling based on mathematical equations as models can be found in (Sadnan and Dubey, 2021). This work aims to model power system using equations that represent the behavior and interactions of various components in the power distribution system. Numerous software tools dedicated for power systems were created like RAPSIM (Pöchacker and Elmenreich, 2014), HOMER (Energy, 1993). Despite their powerful features, these are closed tools that must be used as they are (de Durana et al., 2014). This prevents checking the simulator itself for flaws in the design or simulation algorithm.

3 FORMAL APPROACH FOR M&S DAE-BASED SYSTEMS

Our approach focuses on creating a simulator for DAE-based systems. Our contribution aims to provide a proposal that is formally defined and can be used in diverse domains and applications. Applied to LFA, this simulator is able to simulate static power grid, and represents a preliminary for future work on simulating dynamic-structure grids. This proposal is designed around solid bases such as graphs, the DEVS formalism and symbolic computations. The approach we propose consists in three layers, each addressing a different aspect for M&S of DAE-based systems as shown in Figure 3. Each layer builds on rules and structures defined in its underlying layers.

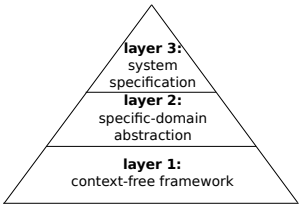

Figure 3: Multi-layer proposed approach for M&S.

3.1 Layer 1 - Context-Free Framework

The context-free framework represents the root layer of our approach. This layer consist in a general simulator architecture and a library of generic DEVS components. It is defined once and for all and is indepen-

dent from the domain of application and the simulated systems. This general simulator architecture can be adapted to various fields by assembling and specializing the generic components in order to perform the specific computations required by the domain.

This architecture, presented in Figure 4, is composed of a clock and three primary blocks: the *data management*, the *mathematical symbols generator* and the *computational engine*.

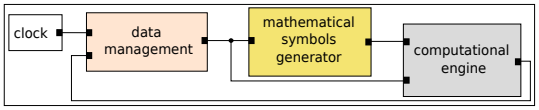

Figure 4: Proposed simulator architecture.

The clock orchestrates the simulation by providing time steps to *data management* block. The *data management* block treats the input data, runs the given scenarios (that represent how changes occur in systems properties over time, these scenarios are specified in the third layer), and prepares the necessary outputs for the next steps in the simulation workflow. The output data of this block is sent to the *mathematical symbols generator* block to generate mathematical symbols that represent system properties. The *computational engine* proceed to computations related to specific needs described into the *specific domain abstraction* layer. Each block has to be specified in the second layer as an assembly of components from the library in order to define a base algorithm for a specific domain. We further detail the library of components available to build each of these blocks in Section 4.

3.2 Layer 2 - Specific Domain Abstraction

This second layer focuses on developing the abstract requirements related to a specific domain. This layer has to be done once for each domain of application we want to support with our approach. This currently has been done for LFA, as will be presented in Section 5.

One aspect of this layer is the definition of a Domain-Specific Modeling Language (DSML) (Blas et al., 2023), which rigorously identifies the main constituents of any given domain. Within this layer, we define the key properties and characteristics of a specific domain to achieve the following objectives:

1. Build a graph-based DSML for the domain.
2. Define the transformation rules that allow to transform graph-based models into equations.
3. Implement an actual simulator for the domain, based on simulator structure defined in first layer.

The graph-based syntax is defined by specifying the nodes and edges properties in order to provide a representation of systems from the target domain that contains all the information required to generate the equations to solve. We also have to specify the transformation rules that guide the conversion from graph syntax to obtain the DAEs that represent the system.

In order to implement a simulator for a given domain, we study the base algorithms that describe the workflow in use within that domain. These algorithms, as well as the transformation rules defined previously, are used to build the simulator. This is done by assembling the generic DEVS components available in the component library and specifying their abstract parameters.

3.3 Layer 3 - System Specification

The third and final layer consists in representing a specific system in the domain of interest, as well as a scenario that represents the system evolution over time. A graph representing the system is injected into the simulator that was built for the domain, along with the scenario. The execution of the scenario over the base system generates the necessary data for the evolution of values directly within the simulator. The simulator then outputs the results in the form of new graphs with updated values at each time step.

4 LIBRARY OF GENERIC COMPONENTS

We have organized the simulator into distinct blocks, each representing a core functionality as represented in Figure 4. To build each block, we provide a library of generic components. The set of components presented here forms a basis, but is not extensive yet. It still needs extension to provide all the required tools that may be necessary for comprehensive support of symbolic computation. For now, only the components required by our target domain were defined.

Each of the components described here is formally specified using the DEVS mathematical syntax. These formal definitions are available in the documentation of our git repository (Attia, 2025). The components also have a concrete implementations that was done using the PyPDEVS simulator (Van Tendeloo and Vangheluwe, 2015) and are available in the demonstrator for the LFA use case, also provided in the same git repository. For the concrete implementation of these components, the symbolic computation part is taken care of by SymPy. Additionally, system's graph representation is carried out

using GraphML (Brandes et al., 2010). In the following, we detail each block of the simulator, and the base components available as of today in the library.

In the *data management* block, the **data generator** component provides input data to the simulator as a graph representing the system and simulation scenarios. The execution result produces input data in the form of a graph with updated property values reflecting system's current state. The **process data** component receives these generated graphs, extracts their nodes and edges, and organizes them into structured data usable by other components.

The *mathematical symbols generator* block includes three components. The **symbols generator** component receives structured data and generates mathematical symbols for the system's properties. The **equations generator** takes structured data and the set of symbols as inputs to generate equations addressing the core computations of a particular domain. The **mapping symbols values** component generates, from the structured data and received symbols, a set of pairs containing symbols and their associated numerical values.

In the *computational engine* block, the **compute equations** component performs specific calculations needed to solve particular domain equations, often involving iterative algorithms or domain-specific mathematical operations. It substitutes symbols with their concrete values and evaluates equations. The **convergence checker** determines whether calculated values will diverge or converge. If convergence is achieved, the solving process has reached a steady state and iterative computations are stopped. Otherwise, it sends a message to other components to continue iterative computations and adjusts simulation parameters.

Components from the library are generic and thus make use of parameters. The parameters can be values, variables or functions which are initially abstract, meaning that they are not yet specified. These parameters are specified at layer 2 when building the simulator for a specific domain.

All the components, except for the clock, are timeless. From a DEVS perspective, this means that their internal state change (δ_{int}) is carried on either instantly ($t_a = 0$) or never ($t_a = \infty$). In that second case, the state change can only be triggered by an external event (δ_{ext}). From the simulation perspective, this means that the simulated time will never evolve as a result of a component changing its state. Only the clock has a function of time t_a that will make the simulator time evolve. This behavior is dictated by the fact that the components we deal with here are not part of a *system simulation*, they rather *are* the simulator. Of course, working with components whose evolution is done in zero-time can lead to issues such as infinite events being triggered without the time advancing, leading to a stuck simulation. In our case, this is solved by carefully designing the abstract components so that they wait for all their required data to be received before emitting events themselves. As this precaution is taken at the generic component level, the specialized components used in an actual simulator won't have to deal with that issue.

5 APPLICATION: LFA OF POWER SYSTEMS

In this section, we apply our approach to the domain of power systems by building a simulator able to conduct a LFA. The traditional workflow for conducting a LFA is represented in the Figure 5. The process begins with formulating the mathematical structures required for LFA (yellow frame). Then, an iterative process is conducted (gray frame) that corresponds to the Newton-Raphson method (Akram and Ann, 2015). This flowchart will be used as a base to conduct the application.

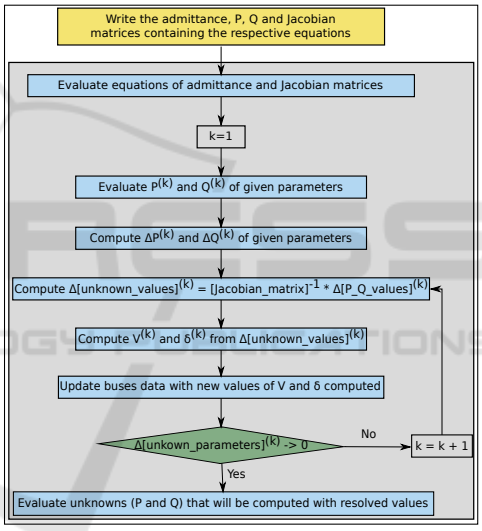

Figure 5: Flowchart of a typical LFA process.

5.1 Power Systems Graph DSML

Using the characteristics of power systems as defined in Section 2.4, we establish the abstract syntax of power systems as graphs. The usual representation of power systems have a quite natural translation to graphs, as illustrated in Figure 6.

We define the syntax of the graph as follows:

- **Nodes (Vertices)** V**:** is the set of buses. Each bus

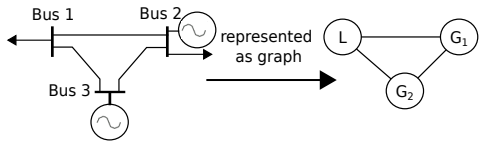

Figure 6: Power system representation with graph.

can contain one or multiple interconnected electrical components (e.g., load, battery, wind turbine, etc.). Each node $v_i \in V$ has several attributes that describe its electrical properties: bus type (bus_type), bus number (bus_no), active power (P), reactive power (Q), voltage magnitude (V), voltage angle (δ). To these intrinsic characteristics, we add the list of *unknown iterative variables* (it_unknown), which refers to the set of variables whose values are initially unknown and have to be determined iteratively. During the analysis, these variables are updated repeatedly using computational algorithm until the solution converges. Each node can be described formally as:

$$v_i = \{\text{bus\_type}, \text{bus\_no}, P, Q, V, \delta, \text{it\_unknown}\}$$

- **Edges E:** is the set of electrical transmission lines that ensures the connection between two nodes. Each edge $e_{ij} \in E$ represents an electrical connection between two buses and has the following attributes: resistance (R), reactance (X), line charging capacity (L), and transformer tap (T).

Formally, an edge between nodes v_i and v_j is described as: $e_{ij} = \{R, X, L, T\}$

- **Mapping function ψ:** this function associates to each edge the pair of nodes it connects. It operates as follows: $\forall\, e_{ij} \in E,\ \psi(e_{ij}) = (v_i, v_j)$

5.2 Graph to Equations Transformation Rules

Based on the LFA workflow (Figure 5), we have to produce a few transformation rules to generate the mathematical objects required to conduct the analysis. As shown in Figure 7, the first rule consists in generating the admittance matrix from graph's edges. The second rule aims to map graph's nodes to generate P and Q equations. Finally, the third rule consist in generating the Jacobian matrix from graph's nodes and the generated P and Q equations.

5.3 Simulator Specification

By following these transformation rules as well as the Newton-Raphson algorithm as defined in Figure 5, we specify a simulator that is represented on Figure 8. We developed this simulator by exploiting the library of generic components described into Section 4.

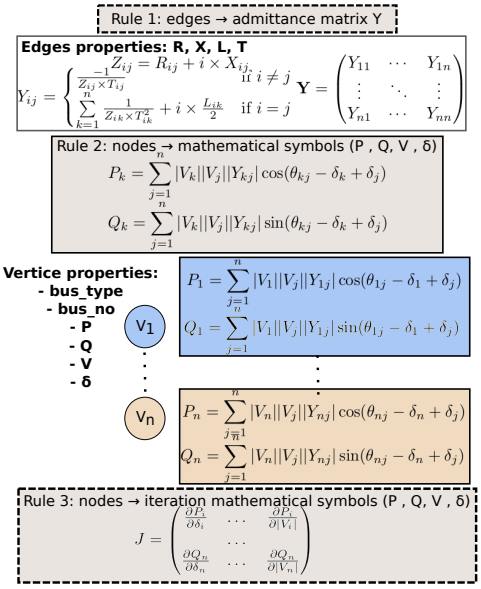

Figure 7: Transformation rules.

The *data management* block generates a graph that represents the system's data. The *mathematical symbols generator* block receives this data and implements the transformation rules. We used the *symbols generator* component to create symbolic representations of the system's essential data, including the voltage, resistance, reactance, P, Q etc. Multiple instances of the *equations generator* component exploit these symbols to generate the admittance, P, Q and Jacobian matrices equations In addition, to provide system's data values we use the *symbols numerical values* component to introduce the corresponding values of defined symbols.

The iterative process is implemented within the *computational engine* block of the simulator. This block instantiate the *compute equations* component that use the data provided by the *mathematical symbols generator* block to compute the admittance, P, Q and Jacobian matrices, and another instance of the same component uses this data to process to the global LFA computations. The iterative process is achieved by an event sent by *convergence checker* to inform the other components if convergence is reached or not.

Once convergence is reached, the system uses yet another instance of *compute equations* to compute the specified P and Q equations that should be computed as results, along with the associated variables, and reinjects the results into the *data management* block.

Using data received from the *computational engine* block and the evolution scenario of the power system, the *data management* block generates a new graph. This graph represents the updated state of the system that can be used in the next simulation step.

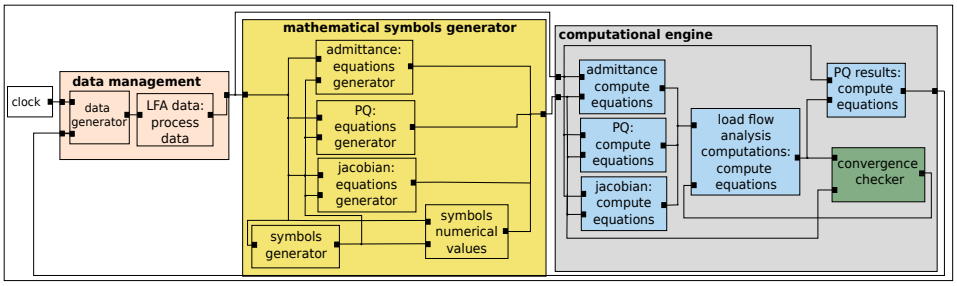

Figure 8: Architecture of Load Flow Analysis simulator.

6 APPLICATION RESULTS AND ANALYSIS

In this section, we present an example of simulated power systems composed of 14 buses. Figure 9 illustrates the architecture of simulated power system represented using the previously defined DSML.

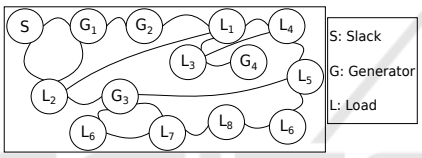

Figure 9: Architecture of simulated power system.

For its evolution, we use a scenario in which we assume that the power generation capacity of the simulated system, represented as *P_generator*, varies throughout the day, as follows:

$$P_{\text{generator}} = \begin{cases} 0 & \text{if } t = 0, \\ \alpha_{\text{init}} + \alpha_1 \times P_{\text{generator}}, & \text{if } 1 \leq t < 7, \\ \alpha_2 \times P_{\text{generator}}, & \text{if } 7 \leq t < 15, \\ \alpha_3 \times P_{\text{generator}}, & \text{if } 15 \leq t < 24. \end{cases}$$

Where $\alpha_{init}, \alpha_1, \alpha_2$ and α_3 are constants used to specify the generator's performance during different periods of the day. The variable t corresponds to the hour within a 24-hour day and is used in the simulator as the simulation time step.

For this system, the following constants define the variations of the *P_generator* parameter: α_{init} = 0.1;0.2;0.05;0.0011 respectively for its four generators. The other constants values are: $\alpha_1 = 1.1$, $\alpha_2 = 1.2$, $\alpha_3 = 0.9$. In these examples, *P_generator* increases during the morning and decreases in the evening, similarly as a photovoltaic production.

By injecting the systems specifications and the scenarios into the LFA simulator, we obtain various results, including the charts as illustrated in Figure 10. In these charts, blue color presents slack's power, green color presents generator's power and other colors present the load's power. The simulation results, according to a specific scenario described in section 5,

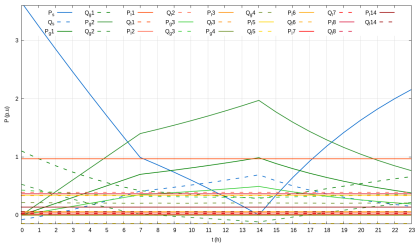

Figure 10: P and Q variations of 14-bus power system.

exhibit the key patterns of power system behavior across different network configurations. For instance, as shown in Figure 10, during daytime (t $\in$ [1,14]) the generated electrical power by the generator (e.g photovoltaic panel) increases. During evening periods (t $\geq$ 15), the generated power decreases. These variations reflect a daily power generation pattern.

The simulation results highlight the slack bus's crucial role in system stability. As generation increases in the morning, slack power (P and Q) decreases. In the evening, as generation decreases, slack power rises to compensate, ensuring balance between consumption and generation.

The simulation results validate key aspects of our proposed approach: the **graph-based representation** successfully captures system topology across configurations, revealing component relationships and temporal evolution; **transformation rules** ensure faithful conversion of graph structures to mathematical equations, properly generating the necessary system representations; and the **modular simulator architecture** demonstrates flexibility in handling different network configurations for DAE-based systems. Out of lack of space, only one example can be provided here. However, various test cases have been defined and are available in our git repository (Attia, 2025).

Although the proposed simulator has been successfully implemented for M&S of LFA, certain aspects require further attention such as the integration of dynamic structure systems. We are now on the process of adding the structure change rules required to make this a complete, dynamic structure simulator. It

will then be able to represent a system and simulate its internal structure changes over time. To address this, we are exploring a graph tool called GMTE (Hannachi et al., 2013) that is able to automatically generate new system structures according to specified rules.

7 CONCLUSION AND FUTURE WORKS

This paper presents a formal approach to build simulators and models for DAE-based systems. Our approach proposes a methodology as well as building blocks that can be used to address simulation in various domains. The proposed methodology is built upon three layers: *context-free framework*, *domain-specific abstraction* and *system specification*.

To demonstrate the practical application of this approach, we implemented a simulator for LFA, modeled power systems with a graph-based DSML. We built the domain-specific abstraction based on LFA methodology, developed transformation rules, and specified the simulator for the domain.

In summary, this work presents a flexible framework for building models and simulators for DAE-based systems. The layered structure is thought with adaptability of the framework to various domains and applications in mind.

Future work will focus on the integration of dynamic structure systems, which is still an ongoing process. Another major work to come is to apply the approach depicted in this article to other domains. Potential candidate domains include power electronics and electric vehicle charging, which represent different scales of power systems.

REFERENCES

Akram, S. and Ann, Q. U. (2015). Newton raphson method. *International Journal of Scientific & Engineering Research*, 6(7):1748–1752.

Albadi, M. and Volkov, K. (2020). Power flow analysis. *Computational Models in Engineering*, pages 67–88.

Attia, A. (2025). Approach to model simulate load flow analysis. https://gitlab.laas.fr/aattia/load_flow_analysis_simulator.

Bickle, A. (2020). *Fundamentals of graph theory*, volume 43. American Mathematical Soc.

Blas, M. J., Gonnet, S., Kim, D., and Zeigler, B. P. (2023). A context-free grammar for generating full classic devs models. In *2023 Winter Simulation Conference (WSC)*, pages 2579–2590. IEEE.

Bouassida Rodriguez, I., Drira, K., Chassot, C., Guennoun, K., and Jmaiel, M. (2010). A rule-driven approach for architectural self adaptation in collaborative activities using graph grammars. *International Journal of Autonomic Computing*, 1(3):226–245.

Brandes, U., Eiglsperger, M., Lerner, J., and Pich, C. (2010). Graph markup language (graphml).

Buchberger, B. (1996). Symbolic computation: Computer algebra and logic. In *Frontiers of Combining Systems: First International Workshop, Munich, March 1996*, pages 193–219. Springer.

Caravantes, J., Sendra, J. R., and Sendra, J. (2019). A maple package for the symbolic computation of drazin inverse matrices with multivariate transcendental functions entries. In *Maple Conference*, pages 156–170. Springer.

Chaturvedi, D. K. (2017). *Modeling and simulation of systems using MATLAB and Simulink*. CRC press.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

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

de Durana, J. M. G., Barambones, O., Kremers, E., and Varga, L. (2014). Agent based modeling of energy networks. *Energy Conversion and Management*, 82:308–319.

Energy, H. (1993). Homer energy: Hybrid optimization of multiple energy resources. Accessed: 2025-01-23.

Hannachi, M. A., Bouassida Rodriguez, I., Drira, K., and Pomares Hernandez, S. E. (2013). Gmte: A tool for graph transformation and exact/inexact graph matching. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 71–80. Springer.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., et al. (2017). Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

Pöchacker, M. and Elmenreich, W. (2014). Rapsim: A simulation tool for microgrids. Accessed: 2025-01-23.

Sadnan, R. and Dubey, A. (2021). Learning optimal power flow solutions using linearized models in power distribution systems. In *2021 IEEE 48th Photovoltaic Specialists Conference (PVSC)*, pages 1586–1590. IEEE.

Trott, M. (2007). *The Mathematica guidebook for symbolics*. Springer Science & Business Media.

Van Tendeloo, Y. and Vangheluwe, H. (2015). PythonPDEVS: a distributed parallel devs simulator. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 91–98.

Weedy, B. M., Cory, B. J., Jenkins, N., Ekanayake, J. B., and Strbac, G. (2012). *Electric power systems*. John Wiley & Sons.

Xu, J. (2013). *Theory and application of graphs*, volume 10. Springer Science & Business Media.

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