



**Report of PFE**

# Validation and Development of PLC Design Tool Based on Modeling and Prolog Logic Reasoning

by

DAI Guohao

Troisième année parcours Systèmes Logiciels

Département Sciences du Numérique

INP - E.N.S.E.E.I.H.T.

August, 2023

**Industrial supervisor:**

Mr. CHEN Lixin

SHANGHAI MEINOLF TECHNOLOGY CO., LTD., Nanjing, CHINA

**School tutor:**

Prof. Guillaume DUPONT

ACADIE - IRIT - INPT ENSEEIHT, Toulouse, FRANCE

# Contents

# Abstract

This report chronicles the author's 22-week internship experience, dedicated to exploring the domain of "Modeling and Reasoning for PLC Design Tools". This report addresses the need for agile manufacturing systems that adapt to changing demands. The research explores a new approach to PLC programming, emphasizing agility and reusability, aligned with Industry 4.0 goals. The research is rooted in harnessing the potential of the IEC 61499 standard's structured representations for defining system components. This approach is bolstered by the integration of ontology, RDF, Prolog, and rule-based reasoning, enhancing information management, integration, and decision-making. The application of these innovative technologies amplifies the agility and responsiveness of PLC development in Industry 4.0.

The report outlines the internship's objectives, methods, and achievements. It initiates with a comprehensive analysis of the IEC 61499 standard, unveiling its untapped potential. Subsequently, the creation and description of an object-oriented model, inspired by this standard, are elaborated upon. The development of a GUI-based automation system description model is presented, with the utilization of engineering design patterns to enhance maintainability. In conclusion, this paper highlights not only the technical outcomes but also the personal growth achieved during the internship. It underscores the vital role of understanding business requirements, effective communication, and the development of soft skills in the realm of software engineering. The immersive exposure to an authentic industry environment provides valuable insights and cultivates a deeper understanding of innovative development strategies. This research signifies a stepping stone toward more advanced and agile PLC development, while also illuminating the path for future exploration and improvements.

*Keywords:* Industry 4.0, IEC 61499, Object-Oriented, RDF, Engineering Pattern

# Acknowledgement

I would like to extend my sincere gratitude to Director CHEN Bo of *SHANGHAI MEINOLF TECHNOLOGY CO., LTD.* for generously affording me the opportunity to embark on an enlightening internship within the company. I am truly thankful for the insightful exposure I gained through this experience.

Furthermore, I want to express my heartfelt appreciation to my supervisor, Mr. CHEN Lixin, for his unwavering support and guidance during my internship journey. His selfless assistance in navigating the intricacies of the industry and technological nuances has been invaluable to me.

Last but not least, I also wish to acknowledge Professor Guillaume DUPONT, my esteemed tutor from the university, whose guidance and assistance in matters related to academic administration have been instrumental in ensuring a seamless learning experience.

DAI Guohao
in Nanjing, China

# Abbreviations and conventions

| | |
|---|---|
| IEC | International Electro-technical Commission |
| XML | Extensible Markup Language |
| RDF | Resource Description Framework |
| PLC | Programmable Logic Controller |
| OO | Object Oriented |
| UML | Unified Modeling Language |
| FB | Function Block |
| OWL | Web Ontology Language |
| GUI | Graphical User Interface |
| IO | Input - Output |
| UUID | Universal Unique Identifier |
| TDD | Test Driven Development |

# 1. Introduction

In the dynamic landscape of Industry 4.0, manufacturing companies are faced with the imperative to remain competitive amidst unpredictable and ever-changing global markets [1]. They are seeking ways to enhance the agility of their manufacturing systems to swiftly adapt to changes in product demands and market requirements. The ability to produce innovative and competitive products hinges on the swift design and deployment of advanced automated production processes. This requires the creation of intricate systems that seamlessly integrate industrial control, manufacturing execution, and business logistics components, culminating in the development of agile manufacturing systems [2]. A pivotal feature of these modern systems is their capacity to swiftly adapt to change, affording manufacturing plants the ability to seamlessly transition between product types and swiftly incorporate new processes to maintain their competitive edge in the market.

The author, DAI Guohao, is engaged in research and development at *SHANGHAI MEINOLF TECHNOLOGY CO., LTD.*, a laboratory automation and inspection company based in Shanghai, China. The company primarily serves heavy industries such as steel, cement, and chemical, providing them with automated testing solutions.

In response to these challenges, there is a burgeoning interest in exploring novel technologies and architectures to cater to the next generation of distributed systems in industrial automation. This paradigm shift in PLC (Programmable Logic Controller) programming seeks to meet the demands of Industry 4.0, characterized by a quest for greater agility, heightened reusability, and a model-based programming approach. To achieve these goals, the proposed approach involves delineating the system's components through the structured representations offered by the IEC 61499 standard. To adeptly manage and store this information, the researchers harness the potential of

ontology and RDF (Resource Description Framework). Additionally, they harness the prowess of Prolog and rule-based reasoning to autonomously deduce the composition of the PLC system.

Embracing this innovative approach, manufacturing companies can leverage the power of IEC 61499's object-oriented concepts and component-based development to navigate the intricacies of PLC programming in the dynamic Industry 4.0 landscape. By seamlessly integrating ontology, RDF, Prolog, and rule-based reasoning, the researchers enhance information storage efficiency, streamline integration processes, and enable autonomous decision-making. Consequently, the overall agility and responsiveness of PLC development are amplified.

This paper presents a detailed account of the author's 22-week internship period, during which the exploration and development of *"Modeling and Reasoning for PLC Design Tools"* were undertaken. Focusing primarily on the creation and description of OO (Object Oriented) models within the project, this effort has been aimed at preparing for the subsequent logical reasoning, enabling the realization of intelligent decision-making capabilities within the system. By introducing object-oriented methodologies, the author has successfully constructed a model capable of efficiently processing and representing relationships among various system components. This forms a robust foundation for the forthcoming chapters, which will address the system's structure and development.

Chapter Two will provide an introduction to the relevant tasks encompassing research design and system development, furnishing readers with an overview of the project's technological aspects for enhanced comprehension.

In Chapter Three, the comprehensive structure of the entire system will be expounded upon. The emphasis will be placed on detailing the methods employed for model creation and description, as well as the development of a human-machine interface

based on these methods. This chapter will progressively unveil the system's design principles and developmental strategies, providing readers with deeper insights. The design of the human-machine interface will facilitate intuitive and efficient interaction between users and the system, thereby offering robust support for workflow optimization and enhancement.

Chapter Four will focus on discussing the achievements and shortcomings encountered during the development process, as well as outlining future directions for the project. Moreover, it will delve into the insights gained from this internship experience. In addition to this, the author will share valuable lessons learned during the internship period, encompassing areas such as effective teamwork, adept problem-solving, and proficient project management.

# 2. Related work

## 2.1 IEC 61499 Standard

IEC 61499 standard, hereinafter referred to as “the standard”, stands as an internationally recognized standard established by the International Electrotechnical Commission (IEC) to delineate the engineering design and implementation of automation systems within distributed control systems. Prior to the inception of the standard, the development of control software for automation systems relied on a set of languages prescribed by the IEC 61131-3 standard. [3] These languages, encompassing Ladder Diagrams (LD), Structured Text (ST), and Sequential Function Charts (SFC), persist as dominant forces in this domain. Despite their extensive utilization, they employ rudimentary abstractions that render the description of intricate systems error-prone, challenging to grasp, and lacking in opportunities for reusability. Regrettably, given that industrial engineers often specialize in specific industrial sectors rather than software engineering, a significant portion of recent advancements in software engineering remains elusive to the majority. Consequently, a pressing demand has emerged for a robust and user-friendly high-level abstract modeling technique catering to industry practitioners.

The notion of harnessing high-level abstractions to confront software complexity gained recognition in the late 1970s through the pioneering efforts of David Parnas, who introduced the concept of abstract interfaces alongside related concepts like information hiding. These innovative ideas later served as the catalyst for the evolution of object-oriented design using object-oriented programming languages. Additionally, standards such as the Unified Modeling Language (UML) came into play, consolidating various OO concepts under a single umbrella. UML serves as a visual language employed to encapsulate the holistic system, spanning from lofty-level requisites down to intricate implementation specifics.

Acknowledging a similar exigency for elevated abstractions within automation, the IEC 61499 standard, which facilitates a component-driven approach for the development of automation software. A cornerstone of the standard is the introduction of function block (FB) programming, an object-oriented programming paradigm tailor-made to articulate diverse components within distributed control systems. These components encompass sensors, actuators, controllers, and more, all treated as FBs replete with inputs, outputs, states, and behaviors. The interplay between these FBs unfolds through events and data, thereby fostering collaborative synergy among distinct elements of the system. [4]

Key features and advantages of the IEC 61499 standard include:

1. *Modularity and Reusability*: The concept of FBs facilitates easier system construction and maintenance while promoting code reuse.
2. *Distributed Performance*: The standard is suitable for distributed control systems, better addressing the requirements of complex systems.
3. *Flexibility*: Interaction between FBs reconfigured based on system needs, enabling system flexibility and adaptability.
4. *Maintainability*: Since the system is divided into multiple function blocks, modifications and maintenance of individual blocks do not impact the overall system's stability.
5. *Cross-Platform and Interoperability*: The application of the IEC 61499 standard can span different hardware platforms and manufacturers, achieving system interoperability.

The IEC 61499 standard has found widespread use in industrial automation and process control, enhancing system flexibility, scalability, and maintainability to adapt to evolving needs and market environments. It provides robust engineering tools and methods for the design and development of distributed control systems.

In today's landscape, many traditional companies in the field of automation are actively developing tools based on the IEC 61499 standard. One notable example is "Eclipse 4diac". [5] Eclipse 4diac is a significant open-source development environment that aligns seamlessly with the IEC 61499 standard, offering comprehensive tools for designing, configuring, and deploying industrial automation systems. This platform empowers engineers to apply the principles of IEC 61499, enabling the creation of modular and adaptable control applications.

### *2.1.1 Models and concepts*

IEC 61499 standard defines several types of models to describe different aspects of distributed control systems. Here are the main model types within IEC 61499 [2]:

(a) *System Model*:

At the physical level, describes the relationship between devices, applications and their interconnections in a distributed system. It includes connectivity and collaboration between devices.

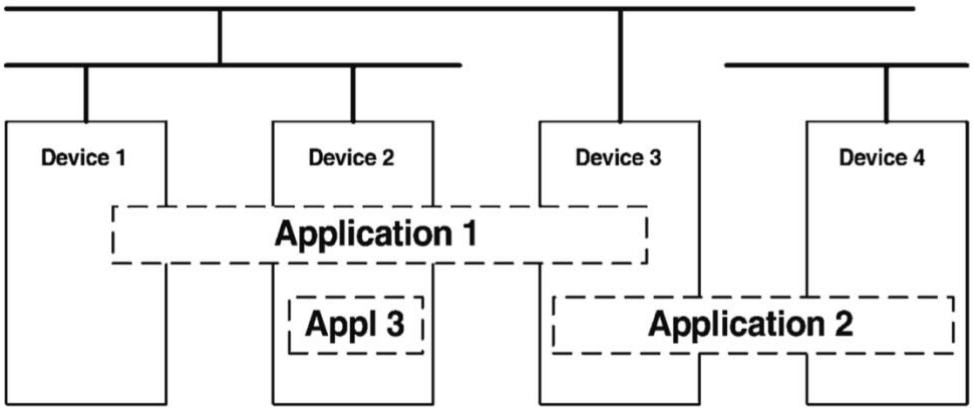


*Figure 2.1.1: IEC 61499 system model*

(b) *Resource Model*:

Describes the functions and services provided by resources to enable the execution of function block applications. Resources allocate function blocks to devices. A

resource furnishes facilities and services to support the execution of one or more function block application fragments. Function blocks in a distributed system are allocated to resources in interconnected devices.

(c) *Application Model*:

Depicts the composition of applications using interconnected FBs, along with the data and event flows between them. Applications can be divided into sub-applications. An application can be fragmented and distributed across multiple resources. Sub-applications further decompose an application, with each sub-application having the characteristics of a function block and potentially spanning other resources.

(d) *Function Block Model*:

Specifies the behavior and internal structure of function blocks. Different types of function blocks define various behaviors and functions. Central to the IEC 61499 architecture is the function block model, described as a "functional unit of software" with its own data structure manipulated by one or more algorithms. A function block type definition outlines the data structure and applicable algorithms. In the upcoming subsection, the fundamental concepts of Function Blocks will be explored.

These model types in the IEC 61499 standard are intertwined to form the overall architecture of distributed control systems. Each model type serves a unique purpose, collectively providing a comprehensive framework for describing different aspects of distributed control systems.

### *2.1.2 Function blocks*

As mentioned earlier, the Function Block (FB) Model serves as the foundational framework for the entire IEC 61499 architecture. Within IEC 61499, various forms of

function blocks are defined, with their primary characteristics summarized as follows [2]:

- Each FB type is identified by a type name and an instance name. These names should always be displayed when visually representing the block.
- Each FB features a set of event inputs, capable of receiving events from other FB via event connections.
- One or more event outputs are present, facilitating the propagation of events to other FB.
- Data inputs are provided, allowing the passage of data values from other FBs.
- Data outputs are designed to transmit data values generated within the FBs to other blocks.
- Each FB is equipped with a set of internal variables, responsible for retaining values between algorithm invocations.
- The behavior of a FB is defined through algorithms and state information. By utilizing block states and state changes, diverse strategies can be formulated to determine which algorithms execute in response to specific events.

Figure 2.1.2 illustrates the fundamental characteristics of an IEC 61499 function block. The upper segment of the function block, referred to as the “Execution Control” section, incorporates a definition, occasionally represented in the form of a state machine, which maps events to algorithms. It specifies the triggering of algorithms defined in the lower segment upon the arrival of various events at the “Execution Control”, as well as when output events are triggered. This causal relationship among event inputs, event outputs, and algorithm execution, as defined by the standard, encompasses means to map the relationships between events arriving at event inputs, internal algorithm execution, and output event triggering.

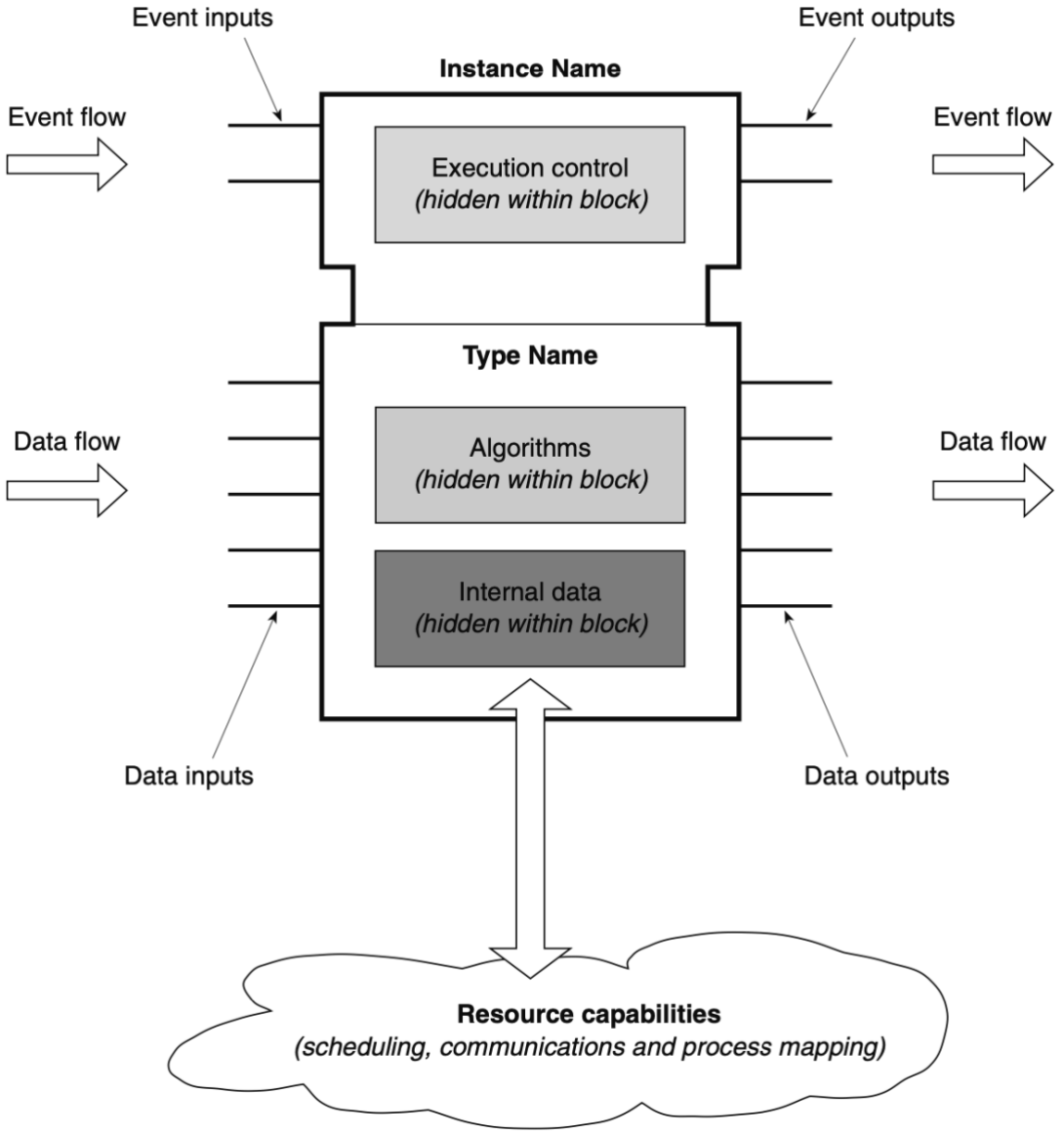


*Figure 2.1.2: Function block characteristics*

The lower segment of the FB contains algorithms and internal data, both discreetly housed within the block. As a software component, FB relies on the support of its encompassing resource to facilitate algorithm scheduling and the mapping of requests to communication and process interfaces.

A function block type consists of a type name, formal definitions for input and output events, as well as input and output variable definitions. It also encompasses the internal behavior of the block, which varies across different block forms:

(a) *Basic function block types*

For basic function block types, behavior is defined through algorithms that respond to input events. As these algorithms execute, they trigger output events, indicating specific state changes within the block. The mapping of events to algorithms is represented using a specialized state transition symbol called an Execution Control Chart (ECC).

(b) *Composite function block types*

In the case of composite function block types and sub-application types, internal behavior is defined by a network of FB instances. This definition also encompasses the data and event connections that need to exist between these internal FB instances.

(c) *Service Interface function block types*

Service Interface function block types introduce an interface between the function block domain and external services. For example, they facilitate communication with FBs devices or the retrieval of hardware real-time clock values.

(d) *Event function blocks*

Event function blocks are a specialized set of function block types designed specifically for handling events. These FBs are intended for modeling and managing event-driven behaviors within distributed control systems. They enable system designers to define how events are triggered, captured, and responded to, allowing for the implementation of complex control logic and interactions.

### *2.1.3 Event Execution Control (ECC)*

A crucial aspect in characterizing the behavior of a FB involves the modeling of the correlation between events and algorithm execution. This task is accomplished through

a concept known as the Execution Control Chart (ECC). The ECC takes the form of a state transition diagram, sharing resemblances with the graphical Sequential Function Chart (SFC) found in IEC 61131-3. Each individual basic FB necessitates an ECC to specify the following aspects:

- The primary internal states of the block. Each ECC must possess an initial state.
- How the block will react to various types of input events. ECCs can incorporate event inputs represented by event variables. Typically, transitions between states are defined by logical expressions employing event input variables
- Each ECC state may encompass zero or more actions. Each action typically corresponds to an algorithm and an associated output event. Whenever a state is active, all actions defined for that state are executed. However, an action might have an empty algorithm, serving solely to trigger an output event. Actions without output events are also admissible.
- The output events that are triggered upon algorithm execution.

## 2.2 RDF and ontology

### *2.2.1 Semantic Web*

The Semantic Web [6] is a concept and framework within the field of computer science and information technology that aims to enhance the understanding and processing of information on the World Wide Web. It extends the capabilities of the traditional web, which mainly focuses on displaying and linking documents, by adding meaning and context to the data. Semantic Web seeks to enable machines and computers to understand the content of web resources in a more intelligent way. It does this by providing a structured and standardized approach to representing and organizing data, making it machine-readable and interpretable. This is achieved through the use of semantic technologies, such as ontologies, metadata, and linked data.

Key components and principles of the Semantic Web include:

(a) *Resource Description Framework (RDF):* RDF is a framework for describing and linking data on the web. It allows information to be represented in a triple format ⟨Subject, Predicate, Object⟩, creating a graph-like structure that machines can navigate and understand.

(b) *Ontologies*: Ontologies define the relationships between concepts, terms, and entities on the web. They provide a shared vocabulary and set of rules for describing domains of knowledge, enabling better data integration and interoperability.

(c) *SPARQL*: SPARQL is a query language used to retrieve and manipulate data stored in RDF format. It allows users to query and analyze data across multiple sources on the Semantic Web.

(d) *Reasoning*: Semantic technologies enable automated reasoning and inference, allowing computers to deduce new information from existing data based on logical rules and relationships defined in ontologies.

In the industrial context, the Semantic Web has significant applications and implications. It enables enhanced data integration, interoperability, and automation in various industrial processes. It offers a powerful framework for enhancing data utilization, integration, and automation in the industrial sector. By enabling machines and systems to communicate, interpret, and process data semantically, it contributes to the realization of more efficient and intelligent industrial processes.

### *2.2.2 XML/RDF*

RDF (Resource Description Framework) [7] constitutes a fundamental technological underpinning within the realm of the semantic web, strategically crafted to expedite the structured representation and exchange of information across the digital expanse. The

architecture of RDF embraces a rudimentary ternary data model, encompassing three constituent components as Notation 2.1:

$$\langle \text{Subject, Predicate, Object} \rangle \tag{2.1}$$

The subject assumes the mantle of the resource undergoing description, while the predicate serves as the conduit for signifying the relational link between the subject and the object. Manifesting as the value or resource intrinsically associated with the predicate, the object consummates the triadic assembly, birthing an interwoven information graph that forms the bedrock of the RDF data structure.

A cardinal hallmark of RDF resides in its adroit utilization of distinct Uniform Resource Identifiers (URIs) to function as beacons for identifying subjects and objects, orchestrating a symphony of connectivity across resources. This judicious approach bequeaths a mantle of global uniqueness and cohesiveness, endowing disparate data sources with the capacity to interconnect with identical resources through a standardized conduit.

In tandem, RDF begets lexicons and ontologies via the deployment of RDF Schema (RDFS) and the Web Ontology Language (OWL). This augmented semantic stratum synergistically enhances the representation of RDF data by affording developers the prerogative to demarcate relationships, classes, and attributes, thereby engendering a landscape conducive to the elucidation of intricate knowledge paradigms and the exercise of refined reasoning faculties. As an essential harbinger of the Semantic Web's grand vision, RDF not only renders information human-accessible but, more crucially, imparts the cognitive faculties to computers for apprehending and manipulating such information. Through its prowess in encapsulating the semantics underpinning relationships and knowledge, RDF bequeaths unto the digital realm a tapestry of interconnected, semantically-enriched data, thus laying the bedrock for avant-garde applications spanning diverse domains.

XML, on the other hand, functions as a versatile markup language that facilitates the structured representation of data. It excels in defining hierarchies, organizing information through tags, and enabling extensible data formats. XML is adept at encapsulating data within user-defined structures, thereby fostering flexibility and modularity in data presentation.

The synergy between RDF and XML is palpable, with RDF often employed to enrich the semantics of data represented in XML. This symbiotic relationship empowers the creation of RDF triples within XML documents, thus endowing the information with enriched context and meaning. By embedding RDF constructs within XML, it becomes possible to seamlessly interlink disparate pieces of information, transforming raw data into interconnected knowledge networks.

### *2.2.3 Ontology*

Ontology, within the context of the semantic web, serves as a cornerstone for structuring and organizing knowledge in a systematic and coherent manner. It can be envisioned as a structured representation of concepts, entities, and the relationships between them within a specific domain. The aim of ontology is to capture the inherent meaning and semantics of information, thereby facilitating effective data integration, sharing, and reasoning.

Ontology provides a formal and expressive framework for describing the various entities and their attributes within a given domain. It enables the establishment of well-defined classes, properties, and relationships that accurately reflect the real-world entities and their associations. By doing so, ontology transcends the mere superficial representation of data and instead delves into the underlying essence and interconnections that govern a particular domain. They are typically constructed using languages such as OWL and RDFS, which offer rich vocabularies and constructs for defining classes, subclasses, properties, and instances. These languages empower

developers to create intricate models that encapsulate the intricate nuances and intricacies of a domain.

Ontology can be leveraged for a multitude of purposes within the semantic web ecosystem. They play a pivotal role in data integration by enabling the seamless aggregation of information from diverse sources. They empower advanced query and search capabilities by enabling the specification of complex relationships and criteria. Ontologies also underpin automated reasoning, allowing systems to derive new knowledge based on the established relationships and axioms.

## 2.3 Prolog

### *2.3.1 Overview*

Prolog, an abbreviation of “Programming in Logic”, is a declarative programming language deeply rooted in formal logic and artificial intelligence research. It occupies a unique niche in the realm of programming languages, characterized by its exceptional paradigm that places paramount importance on logical inference and pattern matching, contrasting with the conventional approach of procedural execution.

In the realm of Prolog, programs are constructed as sequences of predicates, which fundamentally serve as logical assertions delineating relationships among diverse entities. These predicates are systematically structured into rules and facts, thereby constituting a knowledge base that serves as the bedrock for reasoning and computation. The essence of Prolog’s prowess resides in its adeptness at conducting “unification”, an intricate process wherein predicates are aligned to establish logical connections, consequently deducing solutions. Thus, Prolog fundamentally embodies a logical approach to resolving complex problems, epitomized by the formulation of programs as clusters of logical statements known as “clauses”. These clauses intricately define relationships, facts, and rules, all of which encapsulate domain-specific knowledge. The

core methodology of the language centers around the act of querying this knowledge repository to infer resolutions for intricate problems. [8]

The execution modality inherent to Prolog involves a strategic maneuver termed "backtracking". As queries are presented to the system, Prolog diligently endeavors to unearth solutions by systematically scrutinizing the clauses within its knowledge cache. Upon identifying a solution, Prolog promptly presents the outcome to the user. In instances where a figurative impasse is encountered, the system promptly retraces its steps, venturing into alternate routes, thereby engendering a thorough exploration of every feasible solution.

### *2.3.2 RDF related programming*

SWI-Prolog [9] is a fast, robust, and open-source Prolog system that offers comprehensive support for XML and RDF. It not only serves as a powerful tool for logic programming but also provides extensive capabilities for knowledge representation, reasoning, and semantic querying. It provides rich RDF libraries and tools that allow users to load, query, and manipulate RDF data. Users can create, query, and infer RDF triples using Prolog syntax, facilitating efficient management of knowledge graphs. Built upon Prolog's logical reasoning mechanisms, SWI-Prolog enables users to perform inference using RDF triples, thereby deriving new relationships and implicit knowledge. It supports the integration of RDFS and OWL ontologies into the Prolog environment, enhancing both knowledge representation and reasoning capabilities. This seamless integration of logic programming and semantic technologies empowers users to harness the power of Prolog for sophisticated knowledge-based applications and semantic reasoning tasks.

# 3. System design

## 3.1 Requirement

The current project is situated in its initial phase, during which the foremost attention of the investigator is directed towards the formulation of the model and a systematic inquiry into the methodologies pertinent to its delineation. Antecedent to the inception of the domain model's architectural design, a judicious and comprehensive analysis of the exigencies underlying the system is essential:

1. The process commences with the comprehensive curation and diligent scrutiny of pertinent documentation, specifications, and existing systems to foster a profound comprehension. Engaging in meaningful conversations with domain experts and stakeholders is crucial to grasp the foundational principles, components, functions, and intricate interactions within the electrical system. Gathering industry requirements, expectations, and feedback ensures a comprehensive consideration of all pivotal dimensions during the requirement analysis.

2. The substantive phase pertains to the structuring of models encapsulating elements and products ingrained within pragmatic corporate milieus through the medium of an object-oriented paradigm. This encompasses the discernment and depiction of these abstract entities, entailing an in-depth comprehension of the discrete constituents endemic to the electrical system, their functional comportment, interrelationships, as well as plausible utilization scenarios. Guided by the proclivities of the requirement analysis, the demarcation of classes and instances ensues, each class meticulously endowed with attributes and methods. A preliminary class diagram is architected, affording a graphical exposition of interclass relationships, encompassing inheritances, associations, and the like, predicated upon the identified conceptual affinities.

3. The progressive iterations of code formulation and refinement transpire concomitantly with the rigorous regimen of test-driven development. This procedural protocol necessitates a perennial confluence with real-world deployment scenarios intrinsic to industrial applications, thereby fomenting a nuanced and iterative augmentation of class designations, class diagram augmentations, and concomitant augmentations.

4. The iterative refinements thus effected attain a culmination through the comprehensive validation and verification conduits that are forged in conjunction with the insights of domain cognoscenti and stakeholders. Ensuring a faithful correspondence between the model's rendition and the ontological dictates of the electrical system, this phase is substantiated by robust system architecture, judicious coding methodologies and scrupulous version control. A consistent and coherent modus operandi for monitoring and managing requisites ensues, ensuring continual alignment with shifting exigencies whilst corroborating fulfillment. This presumes the sustenance of an astute and receptive dialogue with the project cohort and stakeholder consortium, affording a seamless trajectory for requirement modifications and refinements consonant with project deliverables.

5. Upon completion of the foundational lower-level system model description, the focus shifts to implementing a Graphical User Interface (GUI) tool. This tool serves the purpose of managing modifications to the model's description, enabling users without coding expertise to effortlessly navigate the intricacies of the model system and achieve the desired electrical system design. Adhering to a unified coding style characterized by high cohesion, low coupling, inversion of control, and a steadfast commitment to object-oriented principles is imperative even in the realm of GUI development. The distinct positioning of the GUI at the system's periphery and its distinctive output format render testing a formidable endeavor, thus underscoring the indispensability of rigorous testing practices.

## 3.2 Architecture of domain model

### *3.2.1 Model design*

Through comprehensive analysis of industry requisites and referencing established system specifications (such as the standard IEC 61499), the project lead engaged in collaborative efforts with domain experts to formulate a meticulous hierarchical portrayal of the system—encompassing the “Container”, “Component”, and “Part” strata. The approach embarked upon a methodical bottom-up elucidation of each tier, advancing as follows:

The foundational tier, known as the “Part”, constitutes the elemental bedrock of the system’s architecture, steadfastly immune to compositional amalgamation with elements from the remaining tiers. In essence, the “Part” embodies the elemental constructs from which the system’s rudimentary functions are synthesized. Each discrete “Part” is assigned a distinct function or sub-task, embodying entities encompassing hardware apparatus, sub-circuits, sensors, and actuators. The “Part” tier, assuming the mantle of tangible functionalization, harmoniously interlaces with fellow “Parts”, orchestrating inter-participatory symbiosis. Pertinent interactions with the external landscape are facilitated through Input-Output (IO) entities.

Ascending the hierarchy, the “Component” emerges, an aggregate entity comprising both “Parts” and “Components” (referred to as *child* within the coding paradigm). This amalgam is mathematically characterized as Notation 3.1:

$$\text{Component} \subseteq \text{Component} \uplus \text{Part} \tag{3.1}$$

The “Component” tier orchestrates and coordinates internal constituents, harmonizing communication and cooperation among its affiliated entities (*child*). Concurrently, it governs the intricacies of intra-system and extra-system interactions. Inputs traverse IO entities for assimilation, while outputs are channeled through these entities for dissemination across external interfaces.

Dominating the hierarchy, the “Container” takes center stage, a comprehensive entity capable of aggregating components in diverse configurations. Symbolically expressed through set-theoretical Notation 3.2:

$$\text{Container} \subseteq \text{Container} \uplus \text{Component} \uplus \text{Part} \tag{3.2}$$

The “Container” assumes the responsibility of partitioning the system into coherent and resource-bound enclaves. Each “Container” accommodates a collective assemblage of interconnected “Components” and “Parts”. This architectural paradigm imparts an insular environment, permitting autonomous operation within distinct containers to avert any interference. Notably, IO entities for external interactions are omitted, symbolizing the introspective encapsulation of the “Container”.

In the context of the aforementioned definitions, a pivotal emphasis is placed on the presence of IO entities, constituting a fundamental tenet that governs all interactions beyond the confines of the current entity. This regulatory framework engenders a bifurcation encapsulating the ensuing distinctions:

- IO entities are functionally categorized into two distinct classes: inputs and outputs. Inputs facilitate the acquisition of information from the external realm, while outputs effectuate the dissemination of information to the external domain.

- Further differentiation is based on the type of IO entities, which fall into three classifications: signals (default), electrical quantities, and physical quantities. This classification paradigm is a direct reflection of organizational requisites and aspirations. Notably, it is underpinned by the enterprise’s desire for a discernible demarcation between the electrical component and the physical structure. By delineating these distinct categories and their corresponding units, the system is poised to accommodate the divergent demands of subsequent reasoning and computation, aligning seamlessly with the evolving intricacies of diverse application scenarios.

In the context of the outlined analysis and design, the author progressively refined the code structure through iterative refactoring. The initial step encompassed the formulation of three fundamental abstract classes, which are presented below with enhanced clarity:

As illustrated in Figure 3.2.1, the "*NamedNode*" class was meticulously crafted. This class assumes the role of an abstract representation, encompassing pivotal attributes including an identifier (*id*), a name (*name*), and a type (*type*). Furthermore, it incorporates a set of fundamental methods. For instance, the *toRdfWith()* method assumes the responsibility of transforming node information into the RDF format, subsequently integrating it into the RDF store. By extending the "*NamedNode*" class, subclasses effortlessly inherit the attributes of naming and fundamental properties, thereby mitigating the redundancy associated with multiple definitions

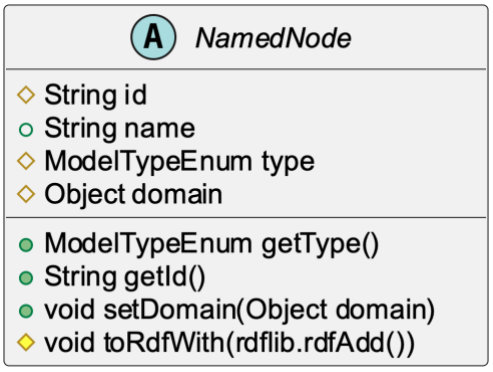


*Figure 3.2.1: UML of NamedNode*

The abstract class "*NodeWithIOs*", shown as Figure 3.2.2, embodies nodes with inherent input and output functionalities. Within its constructor, a private list designated as *ios[]* is instantiated, serving as a repository for IO objects intricately linked to the given node. This class furnishes a range of functionalities, including the addition (*addIO*) and removal (*removeIO*) of IO objects, the systematic iteration through the array of IO objects (*allIOToRDF*), the systematic conversion of IO objects into the RDF

format, and the replication of a new node through the utilization of an existing node as a template

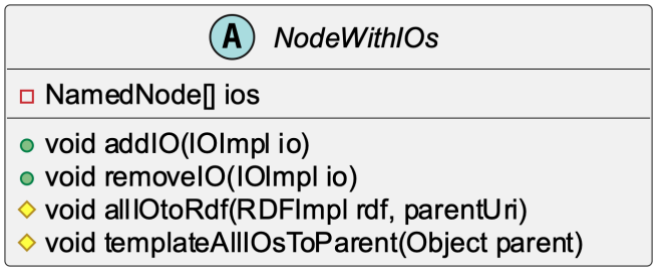


*Figure 3.2.2: UML of NodeWithIOs*

As depicted in Figure 3.2.3, the abstract class "*NodeChildren*" aptly encapsulates nodes encompassing child nodes. Parallel to the "*NodeWithIOs*" class, the "*NodeChildren*" class features the creation of a private list named *children[]* within its constructor. This repository acts as a repository for child node objects intrinsically associated with the parent node. The class provides an array of functionalities including child node addition (*add*), child node removal (*remove*), systematic iteration through the array of child nodes, the coherent conversion of child nodes into the RDF format (*toRDF*), and the systematic templating of nodes. These intrinsic capabilities significantly streamline the management and manipulation of child node information

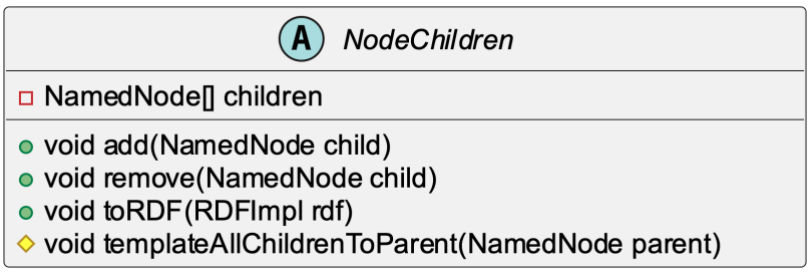


*Figure 3.2.3: UML of NodeChildren*

Building upon the aforementioned abstract classes, it becomes possible to create diverse types of model elements through inheritance. To facilitate the creation of these distinct object elements and to achieve a stable and streamlined codebase, several rounds of code refinement led to the development of a robust factory class design pattern

implementation, "*AutoDescModel*", which serves as a "production" hub for generating the required objects. This approach brings forth numerous advantages, including:

1. *Decoupling*: The factory class effectively decouples the creation and utilization of objects, thereby reducing the level of coupling between different modules. Instead of directly instantiating objects, the factory class is employed to obtain the desired objects, leading to diminished dependencies on specific classes.

2. *Abstracting Concrete Implementations*: Interactions solely involve the factory class, obviating the need to concern oneself with the intricate details of specific product implementations. This protective abstraction safeguards the underlying implementation details of objects.

3. *Code Reusability*: The factory class assumes the responsibility of managing object creation logic, thereby facilitating code reusability. Distinct clients can share the same factory class, eliminating the necessity of duplicating object creation code across various parts of the application.

4. *Control over Instantiation and Lifecycle*: The factory class can exercise controlled instantiation of objects through defined logic, enabling unified management and oversight of object creation and destruction. This capability proves valuable for maintaining a well-organized and monitored lifecycle for objects.

As illustrated in Figure 3.2.4, the classes "*ContainerImpl*", "*ComponentImpl*", "*PartImpl*", "*IOImpl*", "*QtyImpl*", and "*QtyUnitImpl*", which represent model elements, are encapsulated as private inner classes within the "*AutoDescModel*" factory class. These classes can only be instantiated through the corresponding public *create* methods provided by the factory class. This design choice shields the external environment from the intricate implementation details of the inner classes, thereby mitigating the level of interdependency among distinct modules.

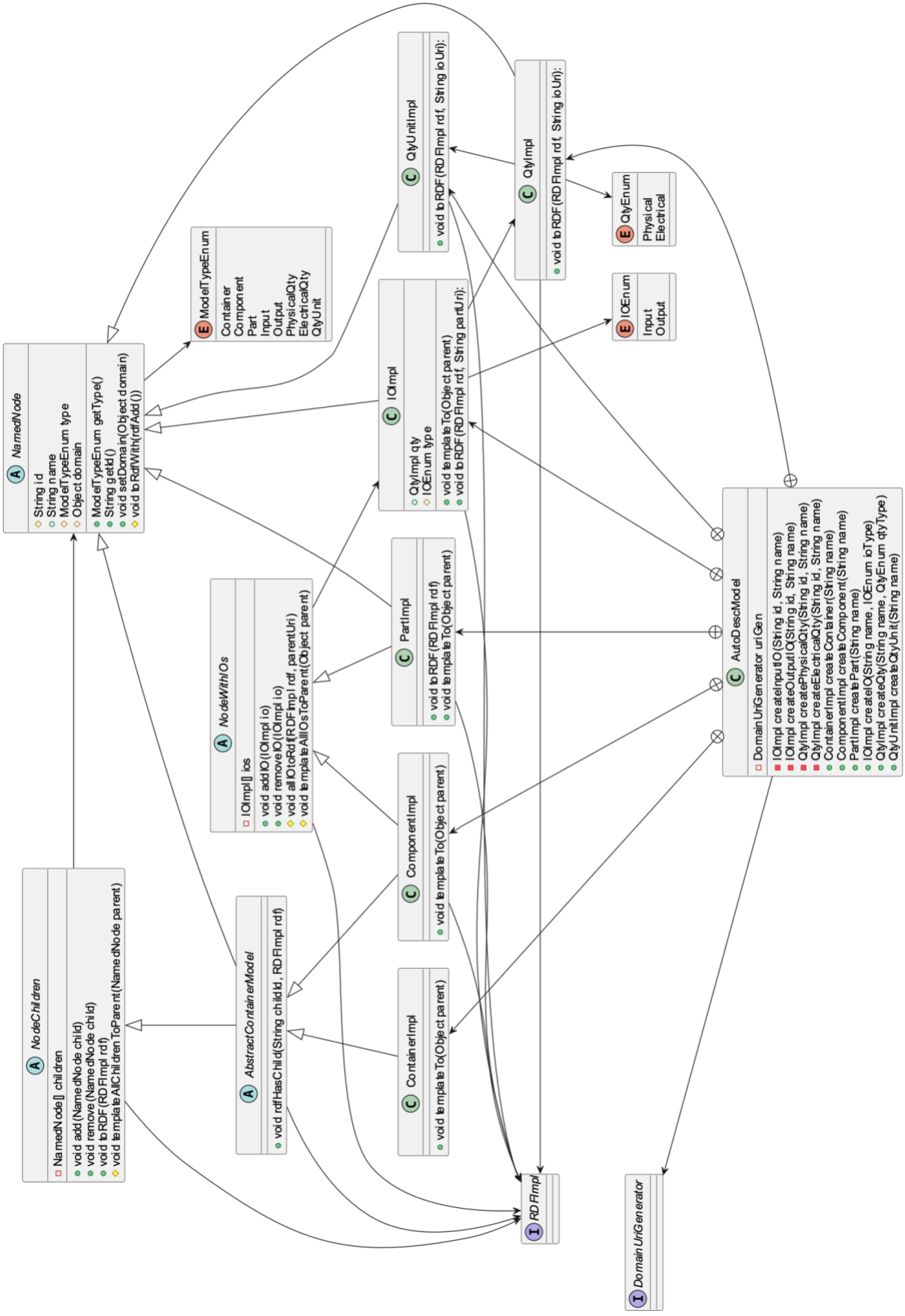


*Figure 3.2.4: UML of AutoDescModel*

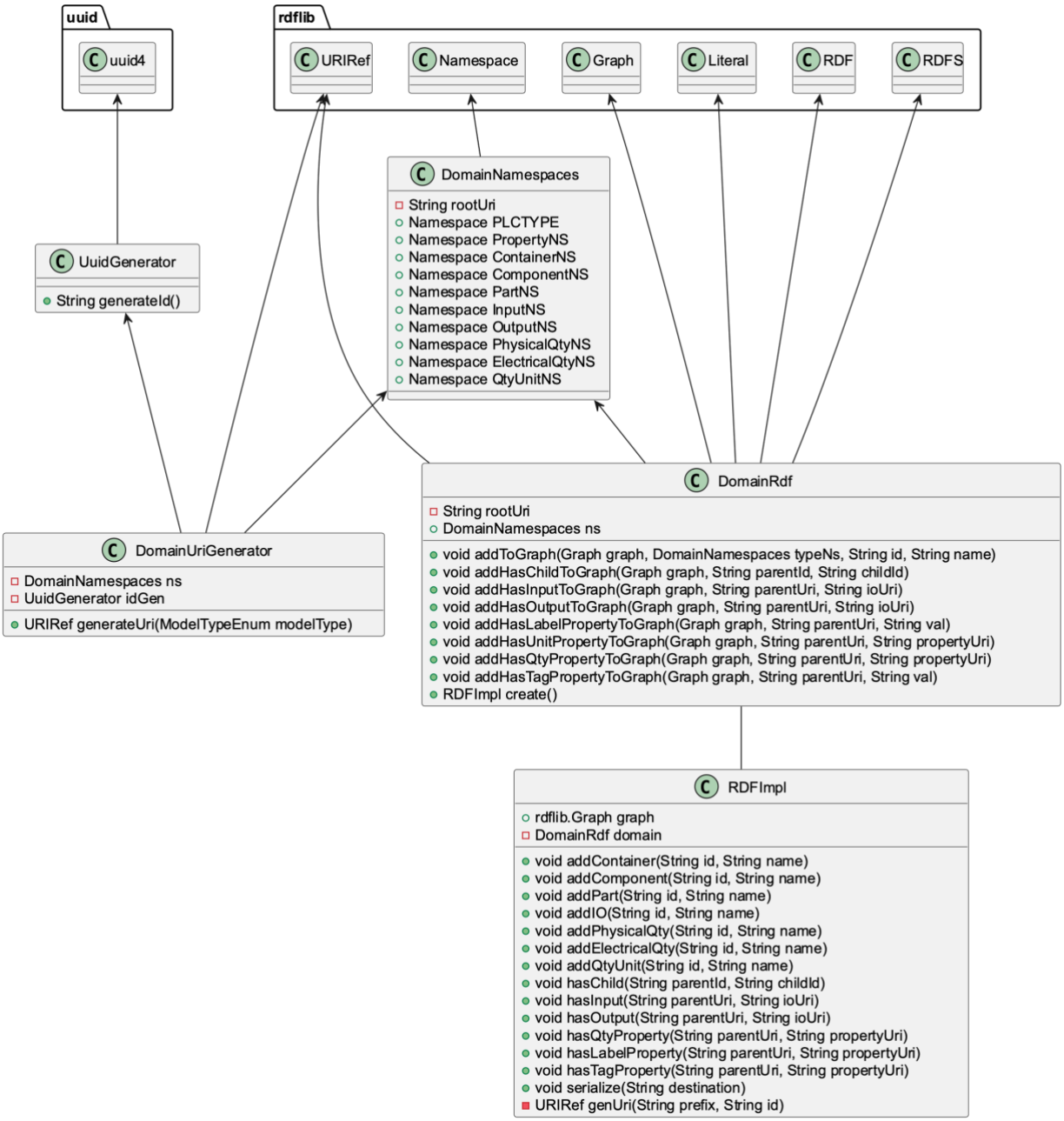


*Figure 3.2.5 UML of rdflib related*

As previously elucidated, the project's utilization of RDF bestows upon it a standardized framework for the meticulous depiction of resources. This framework, in its inherent design, ensures a harmonious blend of machine-readable attributes and human intelligibility. To realize this harmonization, the author has adeptly devised an architecture, as depicted in Figure 3.2.5. Within this architectonic construct, the author ingeniously encapsulated the operational intricacies of generating distinctive identifiers

for object instances by leveraging the *UUID (Universal Unique Identifier)* mechanism, facilitated through the utility class termed "*DomainUriGenerator*". Furthermore, a vital addition to this paradigm is the utility class denominated "*RDFImpl*", diligently engineered to orchestrate the serialization of descriptive model object instances into the RDF format.

### *3.2.2 Test-driven development*

In order to ensure effective Quality Assurance (QA) management, project management, and elevate code quality, the Test-Driven Development (TDD) design pattern has been introduced into the development process. It encompasses the test, driven, and development aspects in software development. In essence, TDD is a holistic approach that intertwines testing, design, and development, providing a structured framework to create reliable and adaptable software systems. In TDD, developers write automated tests for individual program units, where a unit is the smallest testable software component. These tests, often implemented using frameworks like JUnit, can be executed manually or automatically. [10]

Traditionally, unit testing occurred after coding, but TDD reverses this process. Developers write unit tests before coding, allowing immediate test execution. TDD is not solely about testing; it also influences analysis, design, and programming decisions. It integrates testing into analysis and design steps, promoting cleaner code. Refactoring, which modifies code structure without changing behavior, is essential to this process. TDD is a practice rather than a complete methodology. It is intended to aid software construction and can be applied within various process models. TDD yields automated unit tests that remain integral to development. They offer rapid feedback on system changes and assist in maintaining code quality. Although TDD introduces certain challenges, such as maintaining both production code and tests, its benefits include enhanced code quality and an iterative development approach.

Furthermore, the incorporation of *Mock* objects emerges as a salient strategy within this paradigm. This is particularly relevant when a unit undergoing testing is contingent upon modules that are still in the process of development. In scenarios where the unit necessitates the return values of these modules for further processing, *Mock* objects can serve as credible substitutes. This is exemplified through the instance of the *NodeChildren* class:

```
class NodeChildren:

    def __init__(self):
        self.__children = []

    def add(self, child):
        self.__children.append(child)

    def remove(self, child):
        self.__children.remove(child)

    def toRDF(self, rdf):
        for child in self.__children:
            child.toRDF(rdf)

    def children(self):
        for child in self.__children:
            yield child

    def _templateAllChildrenToParent(self, parent):
        for child in self.__children:
            child.templateTo(parent)
```

In the provided illustration, the application of "*unittest.mock*" is employed to establish simulated representations of both the NodeChildren class and its associated subordinate entities. Subsequently, an assessment is conducted on the functionality of the toRDF, children, and *_templateAllChildrenToParent* methods. As shown in the following code[1]:

[1] For further information regarding the mock mechanism in the Python programming language, please refer to the official documentation. [11]

```
import unittest
from unittest.mock import Mock
import NodeChildren

class TestNodeChildren(unittest.TestCase):
    def test_node_children(self):
        # Create mock instances for child objects
        mock_child1 = Mock()
        mock_child2 = Mock()
        mock_children = [mock_child1, mock_child2]
        mock_node_children = NodeChildren()
        mock_node_children._NodeChildren__children = mock_children

        # Mock the toRDF method, verify it is called and check the arguments
        mock_rdf = Mock()
        mock_node_children.toRDF(mock_rdf)
        mock_child1.toRDF.assert_called_once_with(mock_rdf)
        mock_child2.toRDF.assert_called_once_with(mock_rdf)

        # Mock the children method
        mock_child1.toRDF.reset_mock()
        mock_child2.toRDF.reset_mock()
        child_iterator = mock_node_children.children()

        # Verify that the mock method is called and iterate through child
objects
        for mock_child, child in zip(mock_children, child_iterator):
            mock_child.toRDF.assert_not_called()
            self.assertEqual(child, mock_child)

if __name__ == '__main__':
    unittest.main()
```

While the utilization of *Mock* presents several advantageous aspects, it is important to acknowledge that it is not a universal panacea. Within the context of TDD, the application of mocks introduces certain challenges that warrant consideration:

- *Simulation of external dependencies*: When confronted with a profusion of intricate or convoluted dependencies, the process of simulating these dependencies using mocks can become a complex and demanding endeavor.

- *Maintenance of Mock objects*: As the codebase undergoes evolution, the necessity to update mock objects arises. This task can prove time-intensive and susceptible to errors, thereby necessitating meticulous attention.

- *Mocking of private methods*: The act of mocking private methods presents a distinct challenge due to their inherent inaccessibility from external contexts. This can necessitate alternative strategies to facilitate comprehensive testing.

- *Performance implications of Mock*: The extensive deployment of mock objects can potentially impart adverse effects on testing performance. This may manifest as prolonged test execution times and an augmented expenditure of time in completing comprehensive test suites, thus warranting careful consideration.

As such, while mocking offers a valuable tool in the realm of TDD, its deployment demands careful consideration of the aforementioned challenges to ensure its effective and judicious application.

## 3.3 GUI design

To enable non-programming users to easily utilize this model system for achieving the design intent of electrical systems, the author developed a corresponding GUI toolkit using the Qt framework. Furthermore, adhering to a unified coding style of high cohesion, low coupling, inversion of control, and object-oriented principles was emphasized to achieve maintainability. To fulfill this requirement, the author discarded the use of design tools like *QtDesigner* that automatically generate UI code, and instead aimed to encapsulate the used widgets into cohesive classes with singular functionality.

Qt stands as a widely employed C++ application framework, enabling the development of cross-platform desktop, mobile, embedded, and real-time applications. It provides a comprehensive suite of tools and libraries for constructing high-performance, highly customizable, and user-friendly graphical user interfaces (GUI) as well as non-GUI applications. Within Qt, an array of modules and classes is offered for GUI construction, encompassing windows, widgets, layouts, drawing mechanisms, and event handling. Developers can harness these tools to craft intricate user interfaces. Among these, the *Signal/Slot* mechanism stands as a central feature of Qt, enabling loosely coupled communication between objects and enhancing flexibility and maintainability in event handling and interaction.

PyQt, on the other hand, constitutes a GUI framework for the Python programming language, facilitating the creation of rich and interactive desktop applications within the Python environment. Subsequently, a comprehensive exploration of the development process of the GUI toolkit will follow, encompassing both the overarching framework and detailed descriptions of mainly module functionalities.

### *3.3.1 Main window*

As illustrated in Figure 3.3.1, the overall layout of the GUI is presented, consisting of a menu bar, a toolbar, a status bar, and the two delineated areas labeled as ① and ②. Specifically:

- Area ① serves as the display area for the comprehensive hierarchical structure of the project. It is commonly presented in the form of a tree-like directory structure. This component provides users with an intuitive overview of the parallelism or hierarchical relationships among elements within the project. It facilitates the capability for users to interactively select different elements, leading to the presentation of detailed information in Area ②, along with the ability to effect modifications (Distinct display information for elements corresponding to containers, components, and parts is exemplified in Figure 3.3.2).
- Area ②, on the other hand, functions as the container for showcasing in-depth information pertaining to the selected element. This region enables users to engage in modifications of the selected element's attributes and properties.

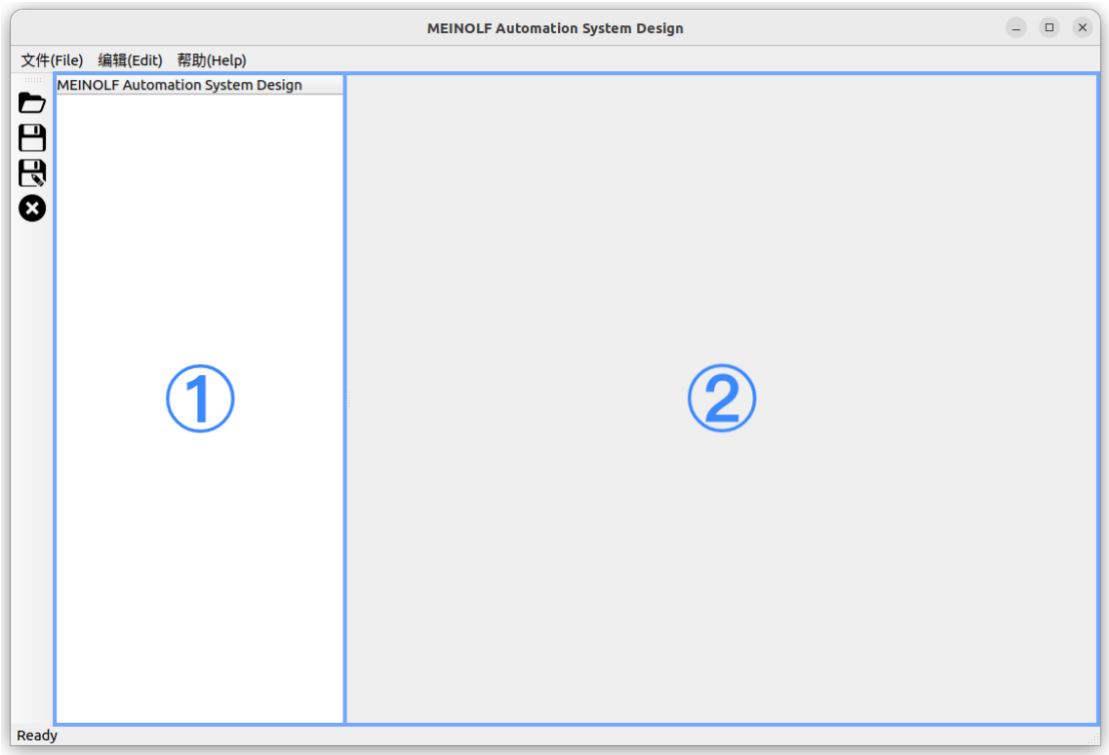


*Figure 3.3.1 GUI's layout*

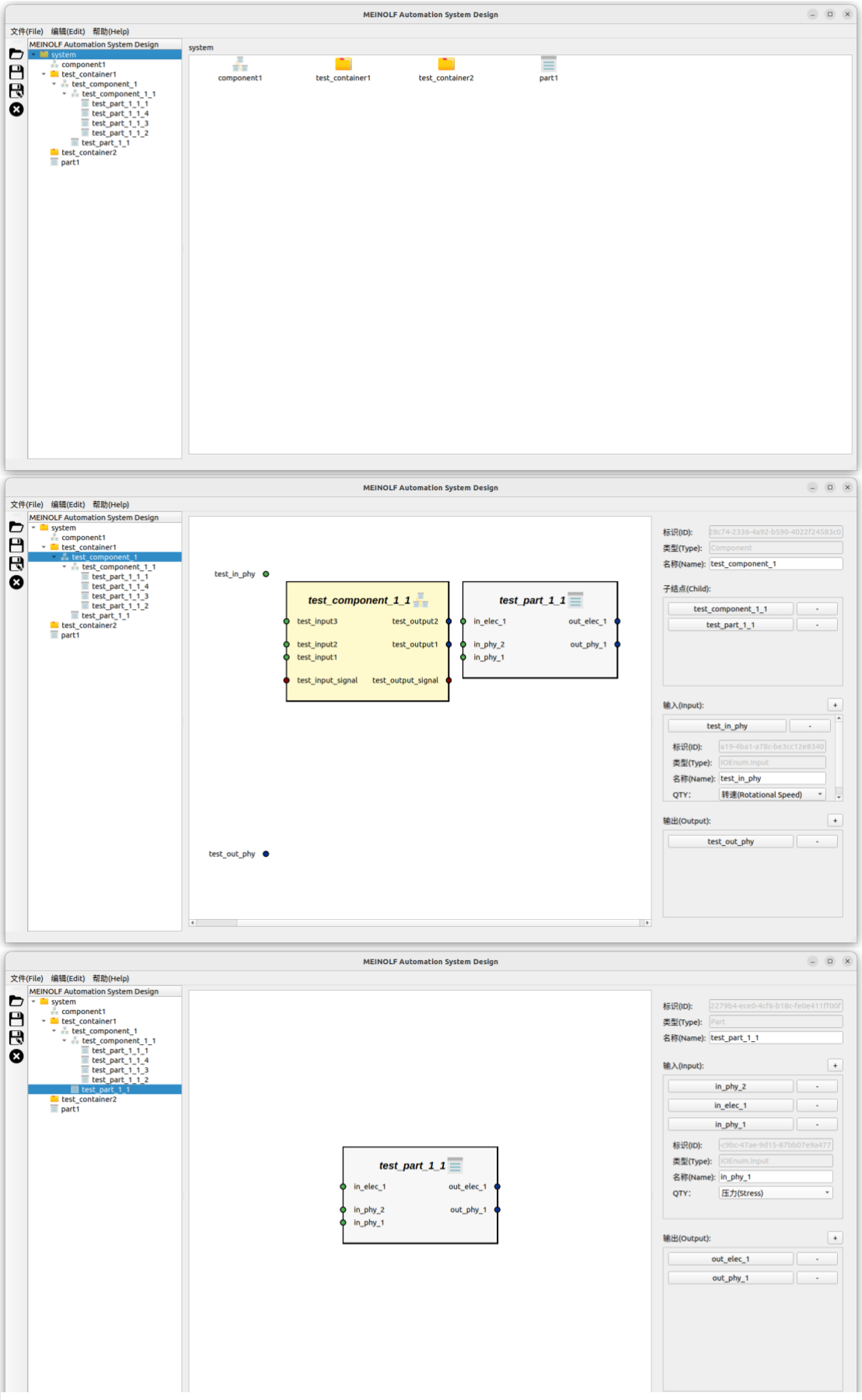


*Figure 3.3.2 containers, components, and parts GUI overview*

### *3.3.2 Actions factory*

Before delving into the comprehensive elucidation of individual GUI elements, it is pertinent to introduce a prevalent design pattern extensively employed during the GUI coding phase: the Factory Pattern. Arising from a particular scenario, the author observed during the coding process that there are numerous occurrences where "an action on a single element affects all elements globally" within the realm of GUI element operations (*Action*). This phenomenon underscores that the lifecycle of *Action* objects, or even the lifecycle of objects upon which *Actions* are contingent, may precede the lifecycle of the objects overseeing the *Actions*.

As an illustrative example, consider a *Container* object *c1* characterized by:

```
Container c1 {
    self.name = "Contain1"
    self.children[] == [Part p1, Part p2]
    ...
}
```

In this scenario, when a modification event (referred to as an "Action") transpires within either Area ① or Area ②, such as the removal of Part p1, it engenders a necessity for concurrent adjustments in both Area ① and Area ②. This dual-sided alteration requirement underscores that modifications initiated within either Area ① or Area ② inherently rely on the presence and behavior of the corresponding GUI objects in both aforementioned areas, thus yielding an interdependent relationship between the modification (*Action*) objects and the GUI objects within both Area ① and Area ② (hereafter referred to as *GUI objects*).

In order to address this challenge effectively, the author chose to employ the Factory Pattern as a solution for decoupling Action objects from GUI objects. This was achieved by incorporating attribute injection, wherein the Action Factory class is integrated into the GUI objects. This approach enables GUI objects to disentangle themselves from the intricacies of Action implementation. Instead, they can interact with Action functionalities through designated function interfaces provided by the Action factory

class. This design choice streamlines the interaction between GUI components and Action functionalities, enhancing the overall modularity and maintainability of the system.

Due to this mechanism, we can effectively centralize the management of actionable Actions for various types of elements, as depicted in Figure 3.3.3.

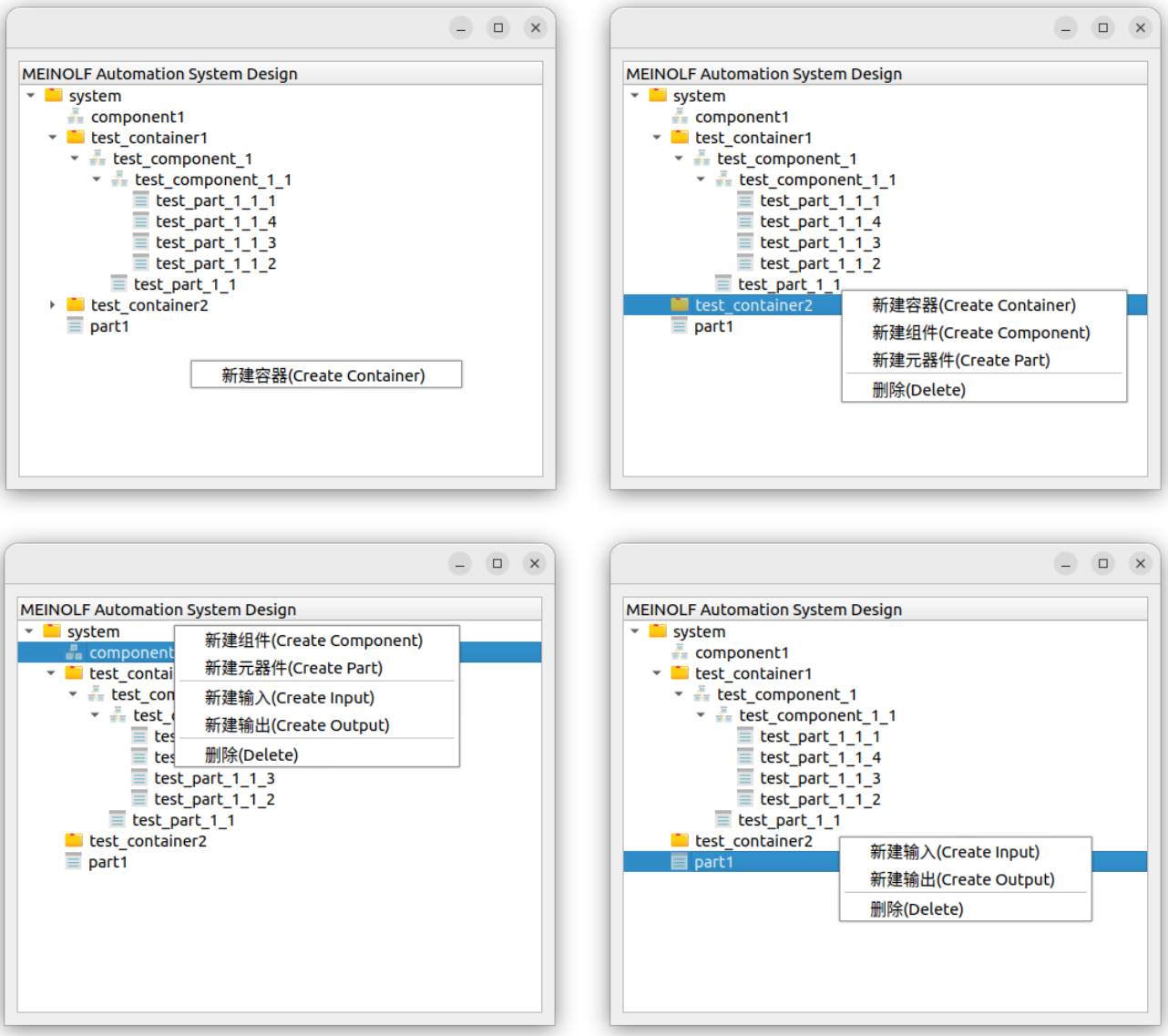


*Figure 3.3.3 The Actions corresponding to different elements (empty node in the top-left, Container in the top-right, Component in the bottom-left, and Part in the bottom-right).*

### *3.3.3 Container editor*

Within the preceding compilation of Figure 3.3.2, an encompassing overview of the Container editor was presented. In the subsequent Figure 3.3.4, a more comprehensive and detailed representation is afforded. In this illustration, the child objects encapsulated within the Container are visually depicted as icons, displayed in the form of icons across the interface.

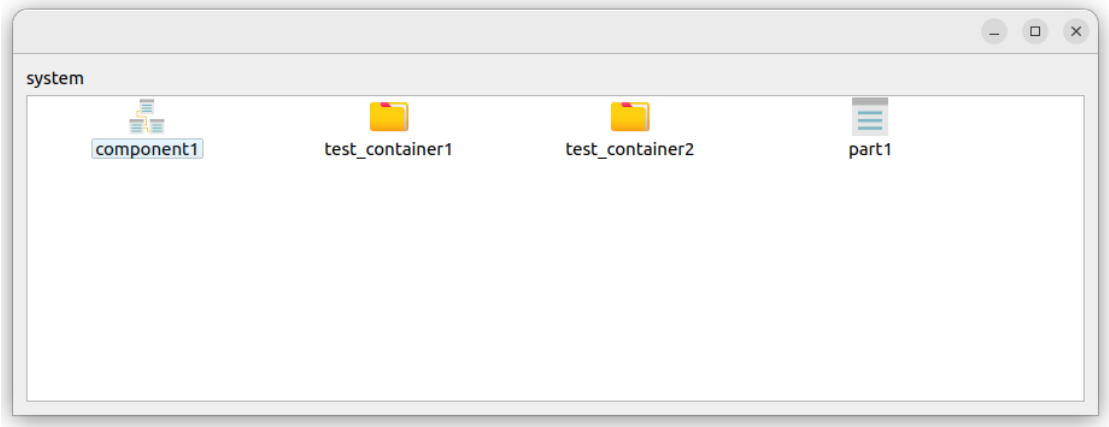


*Figure 3.3.4 Container editor*

### *3.3.4 Part editor*

Regarding a *Part* object, as elucidated in Chapter 3.2, a pivotal emphasis lies on the existence of IO entities. This principle governs all interactions between the current *Part* object and the external environment. Consequently, while employing the GUI for its manipulation, a similar focus is imperative to ensure intuitive and user-friendly operations. Drawing inspiration from the concept of Basic Function Block in the IEC 61499 standard and the foundational framework of the open-source platform Eclipse 4diac™ [5], the author has fashioned the editor for the *Part* object in the configuration depicted in Figure 3.3.5 below. This editor interface is thoughtfully divided into a visually intuitive graphical layout on the left and a panel for attribute modification on the right.

- The left interface leverages the underlying Qt API to craft an abstract representation of a *Part* schematic within the “entity”. This schematic encompasses a rectangular “entity” signifying the *Part*, its designated name, and the encapsulated IO entity objects. All these elements are set against a subdued gray backdrop. Adjacent to the rectangular *Part* “entity” on the left, a range of input types is arranged: the uppermost green dot symbolizes electrical quantities, the middle dot signifies physical quantities, and the lowermost red dot denotes other unidentified or pending attributes. The depiction of output types mirrors the input arrangement, adhering to the same color-coding and layout principles.

- The right-hand interface facilitates the real-time modification of diverse attributes associated with the *Part* object. Notably, any modifications made in this interface are instantaneously mirrored in the left-hand schematic. This synchronization is exemplified in Figure 3.3.6, which portrays the outcome following the deletion of the IO entity “*in_elec_1*”.

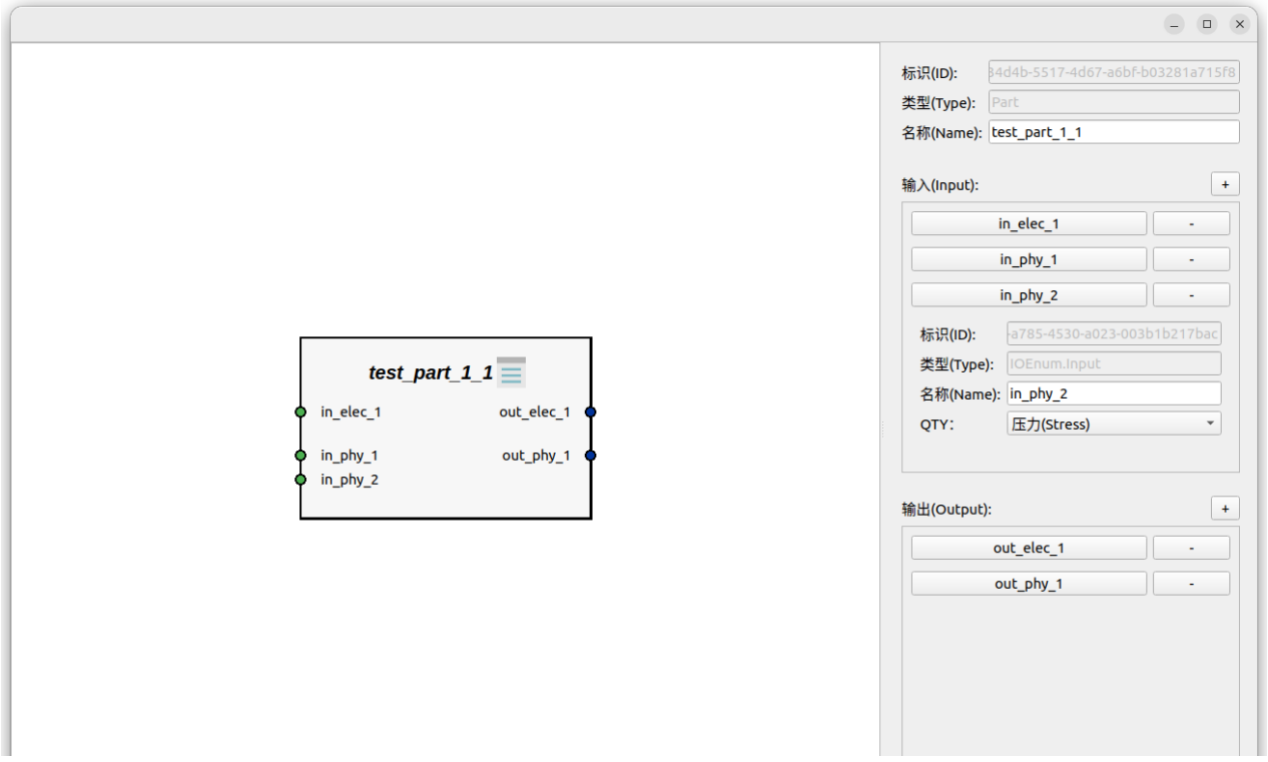


*Figure 3.3.5 Part editor*

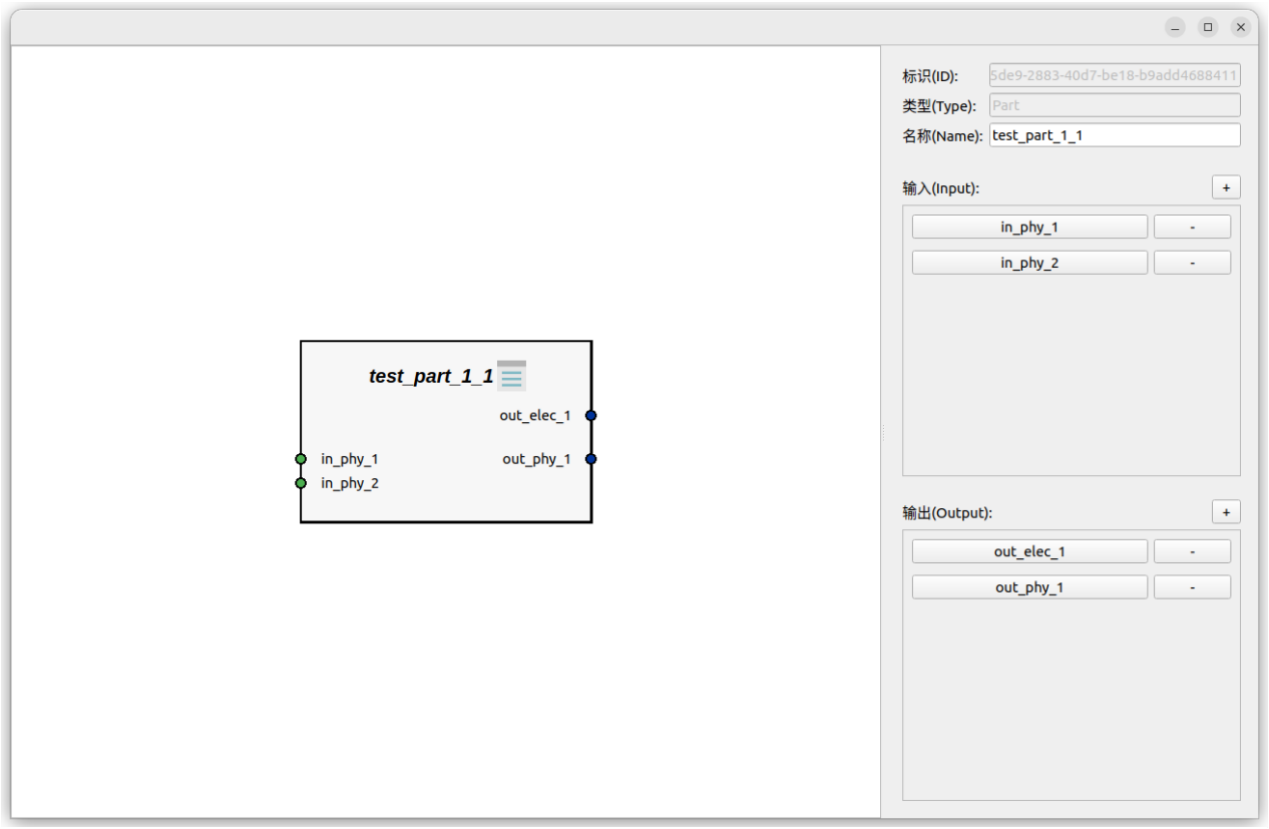


*Figure 3.3.6 Deletion of the IO entity "*in_elec_1*"*

The *QTY* attribute of the *Part* is a selectable value. Users can select a *QTY* instance object from a pop-up dialog using a dropdown Combo Box widget (shown in Figure 3.3.7). This dialog is generated and parses from an RDF resource file located at a designated position during system startup, and it automatically saves the modifications as a file after each alteration. Within the *QTY* editing page, showcased in Figure 3.3.8, the process of selecting *QTY* units aligns with this approach. Figure 3.3.9 and Figure 3.3.10 illustrate the dialog for *QTY* units.

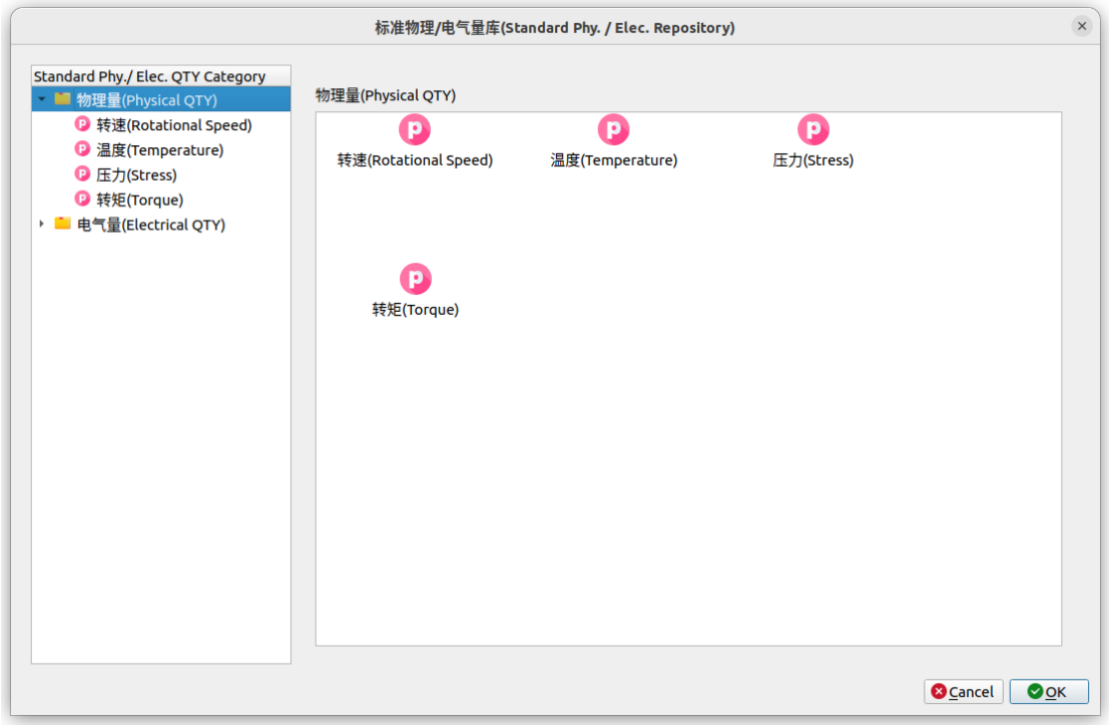


*Figure 3.3.7 Dialog of selecting a QTY*

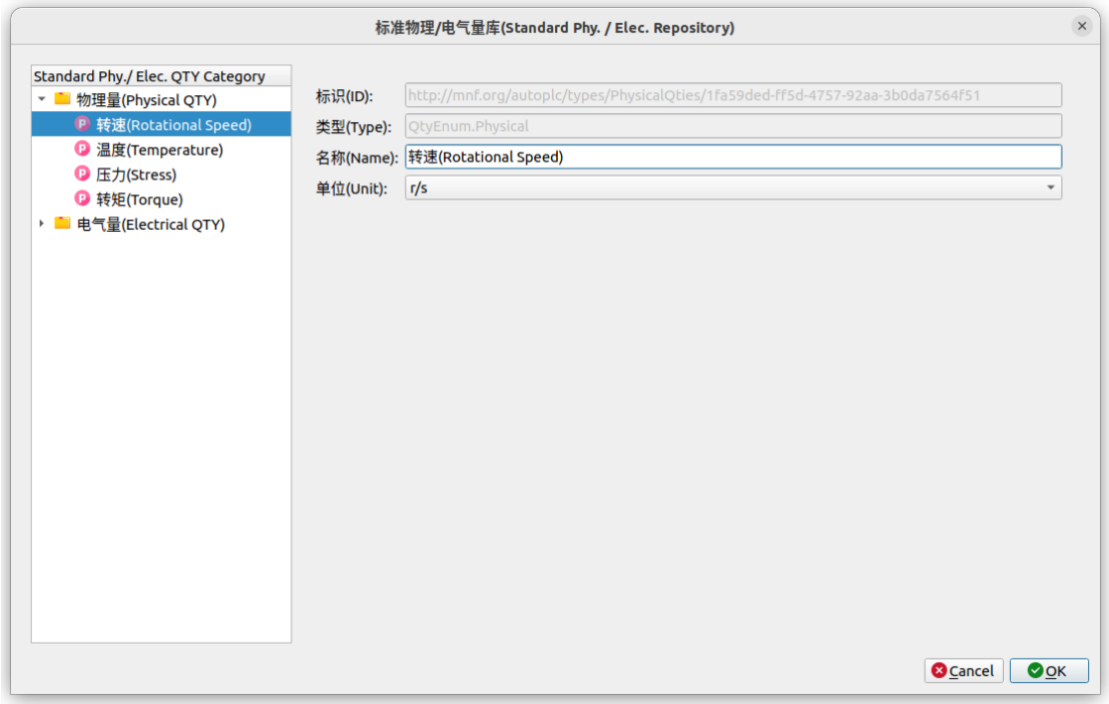


*Figure 3.3.8 Editing a QTY*

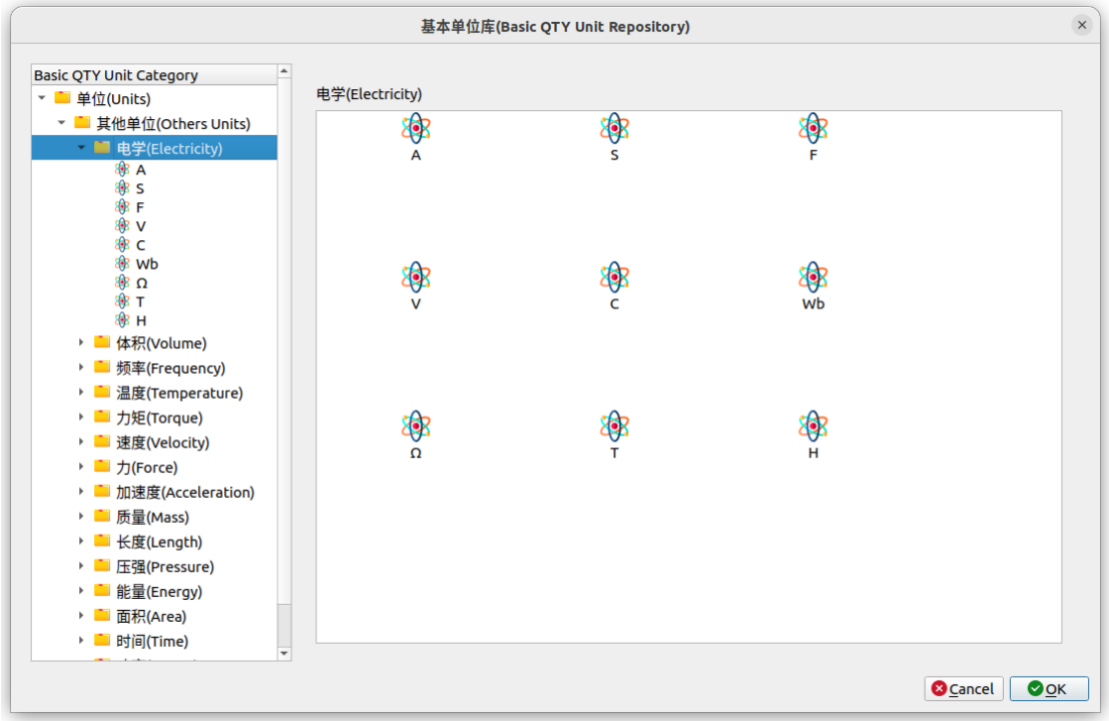


*Figure 3.3.9 Dialog of Selecting a QTY Unit*

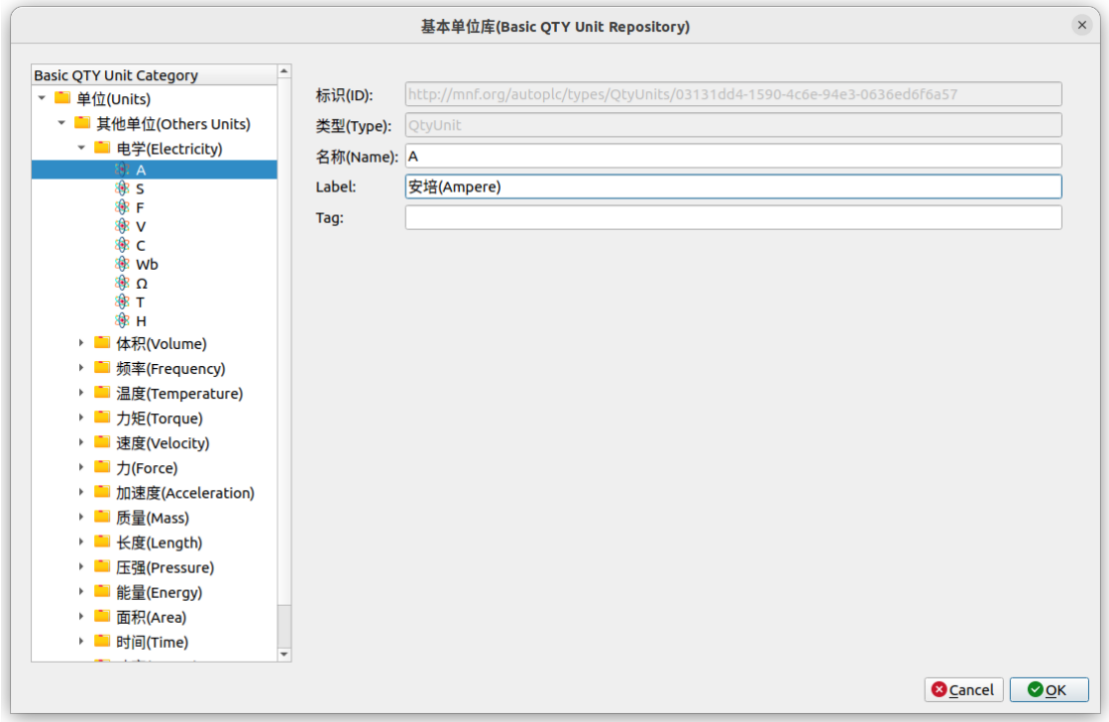


*Figure 3.3.10 Editing a QTY Unit*

### *3.3.5 Component editor*

The *Component*, as described earlier, represents a synthesis of the characteristics found in both *Container* and *Part* objects. Specifically, it combines the presence of child objects along with the ownership of IO entities. To maintain a clear logical structure within the graphical interface, the following guidelines have been established:

*"Within the graphical interface, the Component object focuses exclusively on its immediate offspring. The consideration of child objects' subsequent children is delegated to the child objects themselves, sparing the current object from this concern. Moreover, when a child object assumes the Component classification, its perspective from the parent node – encompassing inputs, outputs, and behaviors – mirrors that of a Part object. To distinguish it, a distinct yellow color scheme is employed (as illustrated in Figure 3.3.11)"*

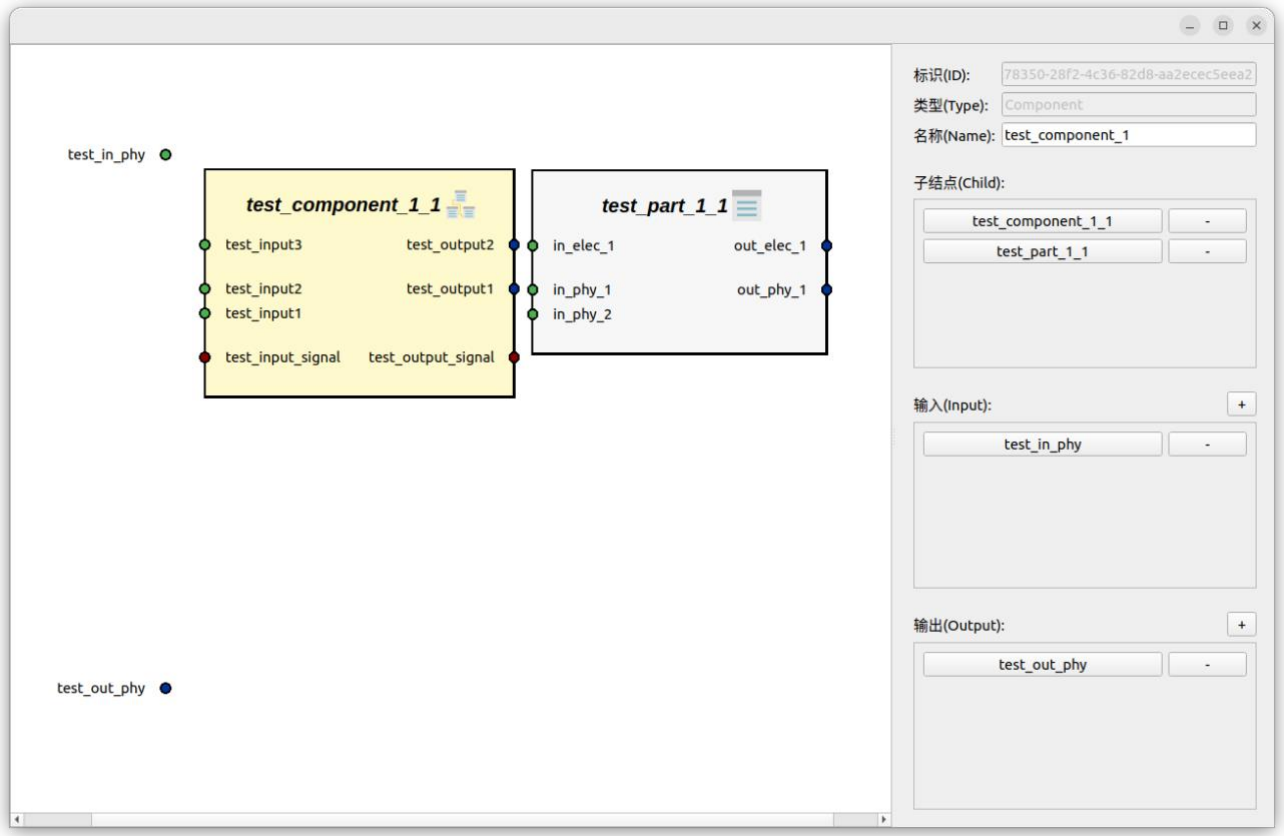


*Figure 3.3.11 Component editor*

Central to the operational efficacy of Part and Component objects are the IO entities, which serve as actionable components. Users have the capacity to orchestrate logical connections by drawing a line (edge) between IO entities, thereby reflecting their intention to establish meaningful connections across the entire system. This is also the direction of the later work.

# 4. Conclusion

This report provides a comprehensive account of the author's 22-week internship period, during which a preliminary exploration and development were undertaken on the matter of "Modeling and Reasoning for PLC Design Tools".

To begin with, a thorough analysis of the IEC 61499 standard was conducted, an emerging standard with significant untapped potential in the industry. This standard served as a foundation for inspiring the later architectural framework of the resource description model. Simultaneously, RDF was chosen as the storage structure for resource descriptions due to its ability to structure data and facilitate machine interpretation of relationships between resources. Prolog was strategically integrated to provide inferential capabilities for the resource description system, aiming to achieve the goal.

Subsequently, a meticulous and methodical delineation of the author's endeavors in the early and intermediate phases of the project was expounded. This involved the development of a GUI-based automation system description model, rooted in the inspiration drawn from the IEC 61499 standard. A series of engineering design patterns were methodically employed to enhance the code's maintainability. Furthermore, an exhaustive suite of development documents, presented in the form of tests, fortified the progressive journey.

Finally, through collaborative engagement with relevant company departments, cooperation facilitated the integration of the company's electrical and mechanical products into the system description model's database. This strategic step set the stage for subsequent inference stages. Upon securing organizational approval, Appendix A of this report encapsulates a simplified RDF representation, aligned with *Figure 3.3.2* of *Chapter 3.2*, with sensitive data expunged.

Notwithstanding these endeavors, the current incarnation of the system description model is not devoid of limitations. Notably, it is presently incapable of capturing inter-element connectivity relationships, restricted to articulating the presence and configuration of system elements. Functionality depiction and internal logic within elements remain elusive. This delineates a promising avenue for forthcoming exploration, albeit detailed exposition is deferred to future investigations.

The internship experience underscored the paramount importance of adeptly comprehending business requisites. In the context of any project or product, value crystallizes when solutions resonate with underlying needs. Throughout this period, sustaining a collective alignment of software objectives across developers, clients, users, and stakeholders emerged as a cornerstone, accentuating the crucial significance of effective communication. This impels developers to cultivate a diverse array of soft skills beyond their technical prowess—an area the author aspires to further enhance. Additionally, the immersive exposure to authentic industry development environments offered novel perspectives, exemplified by the previously introduced concept of test-driven development, signifying valuable takeaways for individual future development.

# Bibliography


[1] R. W. Lewis, “Modeling control systems using IEC 61499”, in *Control Engineering Series No. 59*, ISBN: 978-0-85296-796-6.

[2] T. Terzimehic, M. Wenger, A. Zoitl, A. Bayha, K. Becker, T. Müller and H. Schauerte, “Towards an Industry 4.0 Compliant Control Software Architecture Using IEC 61499 & OPC UA”, in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Limassol, Cyprus, 2017, pp. 1-4, DOI: 10.1109/ETFA.2017.8247718.

[3] International Electrotechnical Commission (2003) International Standard IEC 61131-3: Programmable Controllers - *Part 3: Programming Languages*.

[4] L. H. Yoong, P. S. Roop, Z. E. Bhatti, M. M. Y. Kuo, “Model-Driven Design Using IEC 61499 - A Synchronous Approach for Embedded and Automation Systems”, in *Springer International Publishing Switzerland 2015*, ISBN: 978-3-319-10521-5, DOI: 10.1007/978-3-319-10521-5

[5] Eclipse 4diac™: IEC 61499 Open-Source PLC Framework for Industrial Automation & Control. https://eclipse.dev/4diac/. Accessed 20 Aug 2023.

[6] G. Antoniou, F. Harmelen, A Semantic Web primer - 2nd ed., in *Massachusetts Institute of Technology 2008*, ISBN: 978-0-262-01242-3

[7] F. Manola, E. Miller, B. McBride. RDF primer, in *W3C recommandation*, 2004, 10(1-107) : 6.

[8] J. Wielemaker, Logic programming for knowledge-intensive interactive applications, in *SIKS Dissertation Series No. 2009-10*

[9] SWI-Prolog: https://www.swi-prolog.org. Accessed 20 Aug 2023.

[10] D. Janzen, H. Saiedian, Test-driven development concepts, taxonomy, and future direction[J], in *Computer, 2005, 38(9): 43-50*.

[11] Mock in Python: https://docs.python.org/3/library/unittest.mock.html. Accessed 20 Aug 2023.

# Appendix A: A simplified RDF example

A simplified and abstracted RDF representation corresponding to *Figure 3.3.2* in *Chapter 3.2*, with sensitive information removed:

```
@prefix ComponentNS: <http://mnf.org/autoplc/types/Components/> .
@prefix ContainerNS: <http://mnf.org/autoplc/types/Containers/> .
@prefix ElectricalQtyNS: <http://mnf.org/autoplc/types/ElectricalQties/> .
@prefix InputNS: <http://mnf.org/autoplc/types/Inputs/> .
@prefix OutputNS: <http://mnf.org/autoplc/types/Outputs/> .
@prefix PartNS: <http://mnf.org/autoplc/types/Parts/> .
@prefix PhysicalQtyNS: <http://mnf.org/autoplc/types/PhysicalQties/> .
@prefix QtyUnitNS: <http://mnf.org/autoplc/types/QtyUnits/> .
@prefix mnf: <http://mnf.org/autoplc/types/> .
@prefix ns1: <http://mnf.org/autoplc/properties/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

mnf:Component a rdfs:Class .
mnf:Container a rdfs:Class .
mnf:ElectricalQty a rdfs:Class .
mnf:Input a rdfs:Class .
mnf:Output a rdfs:Class .
mnf:Part a rdfs:Class .
mnf:PhysicalQty a rdfs:Class .
mnf:QtyUnit a rdfs:Class .
ns1:hasChild a rdf:Property .
ns1:hasInput a rdf:Property .
ns1:hasLabel a rdf:Property .
ns1:hasName a rdf:Property .
ns1:hasOutput a rdf:Property .
ns1:hasQty a rdf:Property .
ns1:hasTag a rdf:Property .
ns1:hasUnit a rdf:Property .

ContainerNS:3980a579-1d42-48fc-bf7e-4174a53424d2 a mnf:Container ;
    ns1:hasChild ComponentNS:c96f7b78-2bab-43ca-829a-0cdbbaa716b0,
        ContainerNS:14b68df7-7ece-4913-9d2c-e5400ca25b24,
        ContainerNS:ed679a34-0269-4064-9289-16b7f75d785f,
        PartNS:2b0ae62b-e1d7-487d-b993-156fe1be1dbe ;
    ns1:hasName "system" .
```

```
ComponentNS:c96f7b78-2bab-43ca-829a-0cdbbaa716b0 a mnf:Component ;
    ns1:hasName "component1" .

ComponentNS:dae98208-b647-4475-99d6-3f5aa447afe8 a mnf:Component ;
    ns1:hasChild PartNS:484f53b8-c0c7-463a-b9c2-98b5c049f27f,
        PartNS:9a3d0d2e-7ab8-4ea0-bf18-aa7650e1c6df,
        PartNS:aa1f8839-07ad-4a12-8b45-baba6fd56079,
        PartNS:e3e4d15c-5962-4558-b165-bb5e1d3e753f ;
    ns1:hasInput InputNS:580d8b5d-f276-48b9-868b-056abfed91f6,
        InputNS:ba1d14c9-6f4c-4d43-a40e-8d668cfb3e66,
        InputNS:c6c47614-fd37-47a4-a1c3-31bf9f985088,
        InputNS:dc2d5ab7-9842-4bfd-a1f6-cd4b1b93c8c5 ;
    ns1:hasName "test_component_1_1" ;
    ns1:hasOutput OutputNS:227f2596-9843-41c3-be65-5af3ef50ccdf,
        OutputNS:286eee3b-f390-4ee9-9b3d-16ec093c61ab,
        OutputNS:6f18256a-2462-49b0-8aef-02a015fa5c9d .

ComponentNS:dd228c74-2336-4a92-b590-4022f24583c0 a mnf:Component ;
    ns1:hasChild ComponentNS:dae98208-b647-4475-99d6-3f5aa447afe8,
        PartNS:142279b4-ece0-4cf6-b18c-fe0e411ff00f ;
    ns1:hasInput InputNS:96b6a9de-ea19-4ba1-a78c-be3cc12e8340 ;
    ns1:hasName "test_component_1" ;
    ns1:hasOutput OutputNS:e8d2fbb5-1f66-4c51-9552-91d51f7a481f .

ContainerNS:14b68df7-7ece-4913-9d2c-e5400ca25b24 a mnf:Container ;
    ns1:hasChild ComponentNS:dd228c74-2336-4a92-b590-4022f24583c0 ;
    ns1:hasName "test_container1" .

ContainerNS:ed679a34-0269-4064-9289-16b7f75d785f a mnf:Container ;
    ns1:hasName "test_container2" .

ElectricalQtyNS:49902bc4-be60-45a9-b4b4-860177b5005e a mnf:ElectricalQty ;
    ns1:hasName "电流(Electric Current)" ;
    ns1:hasUnit QtyUnitNS:03131dd4-1590-4c6e-94e3-0636ed6f6a57 .

ElectricalQtyNS:bd081ed3-ff9b-4e46-b44a-7cec71c66231 a mnf:ElectricalQty ;
    ns1:hasName "电压(Voltage)" ;
    ns1:hasUnit QtyUnitNS:231a712f-199b-44dd-9321-7a3784a4bf90 .

InputNS:580d8b5d-f276-48b9-868b-056abfed91f6 a mnf:Input ;
    ns1:hasName "test_input3" ;
    ns1:hasQty ElectricalQtyNS:bd081ed3-ff9b-4e46-b44a-7cec71c66231 .

InputNS:59dd162c-aca2-4f03-910d-a8ecd8e20cc6 a mnf:Input ;
    ns1:hasName "in_phy_2" ;
    ns1:hasQty PhysicalQtyNS:9eab9c21-70da-4981-aa41-d2f63263c970 .
```

```
InputNS:61a0f7f8-b0fc-465b-aa91-55821f9c74e0 a mnf:Input ;
    ns1:hasName "in_elec_1" ;
    ns1:hasQty ElectricalQtyNS:49902bc4-be60-45a9-b4b4-860177b5005e .

InputNS:96b6a9de-ea19-4ba1-a78c-be3cc12e8340 a mnf:Input ;
    ns1:hasName "test_in_phy" ;
    ns1:hasQty PhysicalQtyNS:1fa59ded-ff5d-4757-92aa-3b0da7564f51 .

InputNS:ba1d14c9-6f4c-4d43-a40e-8d668cfb3e66 a mnf:Input ;
    ns1:hasName "test_input_signal" .

InputNS:bddeb9b3-c9bc-47ae-9d15-87bb07e9a477 a mnf:Input ;
    ns1:hasName "in_phy_1" ;
    ns1:hasQty PhysicalQtyNS:9eab9c21-70da-4981-aa41-d2f63263c970 .

InputNS:c6c47614-fd37-47a4-a1c3-31bf9f985088 a mnf:Input ;
    ns1:hasName "test_input2" ;
    ns1:hasQty PhysicalQtyNS:5eedd355-b406-4d5e-9021-1ae1e6cce432 .

InputNS:dc2d5ab7-9842-4bfd-a1f6-cd4b1b93c8c5 a mnf:Input ;
    ns1:hasName "test_input1" ;
    ns1:hasQty PhysicalQtyNS:1fa59ded-ff5d-4757-92aa-3b0da7564f51 .

OutputNS:07ebe0e3-341c-4429-8d94-57107ad29ef5 a mnf:Output ;
    ns1:hasName "out_elec_1" ;
    ns1:hasQty ElectricalQtyNS:6d05f52c-5768-4f92-88b3-5666df616c53 .

OutputNS:227f2596-9843-41c3-be65-5af3ef50ccdf a mnf:Output ;
    ns1:hasName "test_output1" ;
    ns1:hasQty PhysicalQtyNS:9eab9c21-70da-4981-aa41-d2f63263c970 .

OutputNS:286eee3b-f390-4ee9-9b3d-16ec093c61ab a mnf:Output ;
    ns1:hasName "test_output_signal" .

OutputNS:4218f41a-281d-4376-9682-f6ba5767a446 a mnf:Output ;
    ns1:hasName "out_phy_1" ;
    ns1:hasQty PhysicalQtyNS:9eab9c21-70da-4981-aa41-d2f63263c970 .

OutputNS:6f18256a-2462-49b0-8aef-02a015fa5c9d a mnf:Output ;
    ns1:hasName "test_output2" ;
    ns1:hasQty ElectricalQtyNS:6d05f52c-5768-4f92-88b3-5666df616c53 .
```

```
OutputNS:e8d2fbb5-1f66-4c51-9552-91d51f7a481f a mnf:Output ;
    ns1:hasName "test_out_phy" ;
    ns1:hasQty PhysicalQtyNS:1fa59ded-ff5d-4757-92aa-3b0da7564f51 .

PartNS:142279b4-ece0-4cf6-b18c-fe0e411ff00f a mnf:Part ;
    ns1:hasInput InputNS:59dd162c-aca2-4f03-910d-a8ecd8e20cc6,
        InputNS:61a0f7f8-b0fc-465b-aa91-55821f9c74e0,
        InputNS:bddeb9b3-c9bc-47ae-9d15-87bb07e9a477 ;
    ns1:hasName "test_part_1_1" ;
    ns1:hasOutput OutputNS:07ebe0e3-341c-4429-8d94-57107ad29ef5,
        OutputNS:4218f41a-281d-4376-9682-f6ba5767a446 .

PartNS:2b0ae62b-e1d7-487d-b993-156fe1be1dbe a mnf:Part ;
    ns1:hasName "part1" .

PartNS:484f53b8-c0c7-463a-b9c2-98b5c049f27f a mnf:Part ;
    ns1:hasName "test_part_1_1_1" .

PartNS:9a3d0d2e-7ab8-4ea0-bf18-aa7650e1c6df a mnf:Part ;
    ns1:hasName "test_part_1_1_4" .

PartNS:aa1f8839-07ad-4a12-8b45-baba6fd56079 a mnf:Part ;
    ns1:hasName "test_part_1_1_3" .

PartNS:e3e4d15c-5962-4558-b165-bb5e1d3e753f a mnf:Part ;
    ns1:hasName "test_part_1_1_2" .

PhysicalQtyNS:5eedd355-b406-4d5e-9021-1ae1e6cce432 a mnf:PhysicalQty ;
    ns1:hasName "温度(Temperature)" ;
    ns1:hasUnit QtyUnitNS:1bc998b9-fca7-413f-ab89-2ae80dd0c1cc .

QtyUnitNS:03131dd4-1590-4c6e-94e3-0636ed6f6a57 a mnf:QtyUnit ;
    ns1:hasLabel "安培" ;
    ns1:hasName "A" .

QtyUnitNS:03f840db-4522-420d-a8a8-727203aa526e a mnf:QtyUnit ;
    ns1:hasLabel "转每秒(Revolution Per Second)" ;
    ns1:hasName "r/s" .

QtyUnitNS:1bc998b9-fca7-413f-ab89-2ae80dd0c1cc a mnf:QtyUnit ;
    ns1:hasLabel "摄氏温度" ;
    ns1:hasName "°C" .
```

```
QtyUnitNS:231a712f-199b-44dd-9321-7a3784a4bf90 a mnf:QtyUnit ;
    ns1:hasLabel "伏特" ;
    ns1:hasName "V" .

QtyUnitNS:4acb3c09-4e84-4408-a647-16ef3dc35082 a mnf:QtyUnit ;
    ns1:hasLabel "瓦特" ;
    ns1:hasName "W" .

QtyUnitNS:9f9a69f1-4d2c-4971-b0de-e6f8a74f994e a mnf:QtyUnit ;
    ns1:hasLabel "牛顿" ;
    ns1:hasName "N" .

ElectricalQtyNS:6d05f52c-5768-4f92-88b3-5666df616c53 a mnf:ElectricalQty ;
    ns1:hasName "功率(Power)" ;
    ns1:hasUnit QtyUnitNS:4acb3c09-4e84-4408-a647-16ef3dc35082 .

PhysicalQtyNS:1fa59ded-ff5d-4757-92aa-3b0da7564f51 a mnf:PhysicalQty ;
    ns1:hasName "转速(Rotational Speed)" ;
    ns1:hasUnit QtyUnitNS:03f840db-4522-420d-a8a8-727203aa526e .

PhysicalQtyNS:9eab9c21-70da-4981-aa41-d2f63263c970 a mnf:PhysicalQty ;
    ns1:hasName "压力(Stress)" ;
    ns1:hasUnit QtyUnitNS:9f9a69f1-4d2c-4971-b0de-e6f8a74f994e .
```