Journal of Computer Science and Cybernetics, V.42, N.x (2016), 1–
DOI no. 10.15625/1813-9663/22832

TOWARDS A DOMAIN-SPECIFIC LANGUAGE FOR DEVELOPING CONCURRENT APPLICATIONS USING BIP FRAMEWORK

LE KHANH TRINH*, PHAM NGOC HUNG

VNU University of Engineering and Technology,
144 Xuan Thuy Street, Cau Giay Ward, Ha Noi, Viet Nam

Crossref Similarity Check
Powered by iThenticate

Abstract. Concurrent systems play a crucial role in modern software development, enabling applications to handle multiple tasks simultaneously and improve performance by utilizing the full potential of modern hardware. The Behavior-Interaction-Priority (BIP) framework is a model-driven approach for designing and verifying concurrent and distributed systems. To apply BIP in system design, designers first analyze the system's requirements to construct BIP connectors that describe the coordination between the system's components. Based on these connectors, developers create a BIP model for verification and JavaBIP artifacts for implementation, following an exogenous approach. However, constructing the BIP connectors and creating BIP/JavaBIP artifacts requires expertise in formal methods, model-driven design, and concurrent systems. Consequently, the developers might need to invest time and effort to learn the framework and associated tools. In this work, we propose a novel mechanism to reduce developers' manual efforts when applying the BIP framework, including (1) a pseudo-natural language for specifying behavioral constraints without learning the BIP framework and (2) a compiler for automatically generating the BIP/JavaBIP artifacts, which are then used for developing concurrent software following the exogenous approach. We illustrate the feasibility of our approach by applying NaturalBIP to OCCIwareBIP, a framework for designing, validating, and implementing concurrent cloud applications.

Keywords. Behavior-Interaction-Priority, concurrent system, correct-by-construction, domain-specific language, exogenous coordination.

1. INTRODUCTION

Concurrent systems play a pivotal role in modern software development, enabling applications to handle multiple tasks simultaneously and fully leverage the potential of contemporary hardware architectures [1]. However, the design and implementation of correct concurrent systems remain challenging due to the inherent complexity of coordinating multiple interacting components. The Behavior-Interaction-Priority (BIP) framework [2] presents a promising approach for the design and verification of concurrent and distributed systems [3]. BIP adheres to a model-driven, correct-by-construction methodology, allowing designers to specify component interactions and verify system properties before implementation [4].

*Corresponding author.
E-mail addresses: trinhlk@vnu.edu.vn (L. K. Trinh); hungpn@vnu.edu.vn (P. N. Hung).

Despite its formal rigor, the practical adoption of BIP is hindered by several barriers: (1) the need for expertise in formal methods and concurrency theory; (2) the labor-intensive process of manually defining interaction connectors, especially in systems with dynamic or numerous components; and (3) the requirement to maintain dual artifacts, formal BIP models for verification and JavaBIP implementations for execution, increasing development effort and the risk of inconsistency.

To address these challenges, we propose *NaturalBIP*, a domain-specific language (DSL) that enables software designers to specify behavioral constraints using a structured, pseudo-natural syntax without requiring in-depth knowledge of BIP's formalism. NaturalBIP is supported by an automated compiler that translates high-level specifications into both verifiable BIP models and executable JavaBIP artifacts, bridging the gap between informal requirements and formal implementation.

We demonstrate the applicability of NaturalBIP by integrating it with OCCIwareBIP [5], a model-driven framework for designing and implementing concurrent cloud applications. This integration enables the early specification of coordination policies and the automated generation of correct-by-construction runtime components. To evaluate the effectiveness and soundness of our approach, we formulate and answer the following research questions:

- **RQ1**: Can any BIP connector in the BI fragment (i.e., excluding Priority) be expressed in NaturalBIP?
- **RQ2**: What is the expressive power of NaturalBIP in capturing common coordination patterns?
- **RQ3**: Is the NaturalBIP-to-JavaBIP compiler semantically correct?

Our contributions are threefold: (1) NaturalBIP, a template-based DSL with formal semantics, enabling intuitive specification of BIP connectors using ontology-guided natural language patterns. We show that NaturalBIP is expressively complete for the BI fragment of BIP by providing a semantics-preserving translation from NaturalBIP specifications to dual-Horn clauses, which fully characterize BIP interaction models [6]. (2) NaturalBIP Compiler, an automated tool that generates BIP models for verification and JavaBIP artefacts for implementation, ensuring semantic consistency across design and code. We demonstrate that NaturalBIP supports quantified constraints, conditional synchronization, mutual exclusion, and causal dependencies, patterns prevalent in cloud and distributed systems, using structured templates grounded in a formal ontology. (3) Integration with OCCIwareBIP, demonstrating the practical application of the approach in a real-world cloud development framework. We establish correctness by proving that the generated JavaBIP glue code and BIP connectors define equivalent interaction sets. This equivalence ensures that verification results on the BIP model carry over to the implementation.

The rest of the article is structured as follows. The next section presents a comprehensive review of related work. Section 3 introduces the background knowledge necessary for this paper and some frameworks and tools upon which our proposed framework is built. Section 4 delineates the details of our NaturalBIP language and compiler. We started by presenting an overview of the methodology, then the language for specifying requirements and the compiler to generate corresponding artifacts for the concurrent software development using the BIP framework. Section 5 demonstrates the application of our approach through the OCCIwareBIP framework. Finally, Section 6 concludes the paper and discusses future directions.

2. RELATED WORK

2.1. Domain-specific languages and metamodeling

Domain-specific languages (DSLs) are increasingly used in model-driven engineering to bridge the gap between domain experts and formal implementation frameworks. A well-designed DSL enables non-experts to express system requirements using familiar, structured syntax while ensuring semantic precision through formal underpinnings. The design of DSLs often relies on metamodeling, where a formal schema defines the language's syntax and semantics. In the Eclipse Modeling Framework (EMF), for instance, metamodels are expressed in Ecore and used to generate editors, parsers, and code generators.

In this work, NaturalBIP adopts an ontology-driven metamodeling approach, where the NaturalBIP Requirement Ontology (NRO) serves as the semantic foundation. The NRO is built upon a layered architecture: the top-level Behavioral Ontology (BO) defines generic concepts (e.g., *Subject*, *Action*, *State*, and *Finite State Machine (FSM)*), while the Domain-Specific Ontology (DSO) specializes these for cloud systems by mapping OCCIware metamodel elements (e.g., *Kind*, *Mixin*, and *Transition*) to BO classes. This alignment ensures that NaturalBIP specifications are both semantically grounded and domain-relevant.

Unlike rigid boilerplate languages (e.g., RESA [7], K-BRE [8]), which require users to memorize numerous templates, NaturalBIP uses a small set of grammar-based templates derived from the NRO. These templates reduce ambiguity while preserving expressiveness, enabling designers to compose requirements using natural-language-like patterns (e.g., "*For any Peer p, if p executes register, t shall log*"). This approach balances usability and formal rigor, supporting early error detection through ontology-based validation.

2.2. The BIP framework

The Behavior-Interaction-Priority (BIP) framework [2] provides a formal, component-based methodology for constructing concurrent systems in a correct-by-construction manner. BIP separates component behavior (automata) from coordination (interaction connectors) and scheduling (priority), enabling modular design and formal verification. However, practical adoption of BIP faces several challenges: (1) BIP requires expertise in formal methods, particularly in constructing interaction connectors using the Algebra of Connectors (AC) [6]. This limits accessibility for software engineers without a background in concurrency theory; (2) For systems with dynamic or numerous components, defining all possible interactions becomes combinatorially complex and error-prone; and (3) Designers must maintain both a BIP model for verification and a JavaBIP implementation for execution, increasing development effort and the risk of inconsistency.

To mitigate these issues, tools like the BIP Eclipse plugin and Dy-BIP [9] support model construction and dynamic reconfiguration. However, they do not simplify the specification of coordination logic. Our work addresses this gap by introducing a high-level, natural-language-inspired DSL that automates the generation of both BIP and JavaBIP artifacts.

2.3. Formal semantics and verification of BIP models

The formal semantics of BIP are grounded in labeled transition systems and the Algebra of Connectors (AC), which defines interaction models as sets of allowed synchroniza-

tions. AC connectors are characterized by dual-Horn clauses, Boolean formulas of the form $p \Rightarrow a_1 \vee a_2 \vee \cdots \vee a_n$, where p is a port and each a_i is a conjunction of ports. This logical characterization enables both verification and code generation. Verification of BIP models is typically performed using tools like iFinder[1], which employs compositional model checking to combat state-space explosion. iFinder analyzes components in isolation using interaction invariants, logical constraints that capture permissible synchronization patterns. These invariants are derived from the glue logic and encoded into SMT solvers to verify safety properties such as deadlock freedom.

Our approach leverages this formal foundation: NaturalBIP specifications are translated into Boolean formulas, then into dual-Horn clauses, ensuring semantic equivalence between the high-level requirement and the generated BIP/JavaBIP artifacts. This guarantees that verification results on the BIP model remain valid for the JavaBIP implementation, preserving the correct-by-construction guarantee.

Several formalisms have been proposed for exogenous coordination, including REO [10], the Tile Model [11], and nets with boundaries [12]. While these provide rigorous semantics, they lack high-level, intuitive interfaces for requirement specification: *REO* uses channel-based connectors to define data flow between components but requires low-level graphical or textual modeling; the *Tile Model* offers a categorical framework for component composition but is primarily theoretical and not widely adopted in practice; and *nets with boundaries* extend Petri nets for open systems but are less suited for state-dependent coordination. In contrast, *NaturalBIP* combines the expressiveness of formal connector algebras with the usability of natural-language patterns, making it more accessible for real-world cloud application design.

3. BACKGROUND

This section presents the core theoretical and technical foundations essential for understanding the NaturalBIP compiler's design and operation. It focuses on the BIP framework, its Java-based implementation (JavaBIP), the iFinder, and the Boolean formula processing used in the translation pipeline. These elements form the target formalism and toolchain into which NaturalBIP specifications are compiled.

3.1. The BI(P) framework

The Behavior-Interaction-Priority (BIP) framework [2] is a component-based, model-driven approach for constructing concurrent systems in a correct-by-construction manner. It separates system design into three orthogonal layers: Behavior - Components are modeled as Finite State Machines (FSMs) with states and transitions; Interaction - Coordination is defined via connectors that specify allowed synchronizations among component ports; and Priority - Scheduling policies resolve non-determinism by prioritizing certain interactions.

However, this work operates within the BI fragment of BIP, excluding the Priority layer, because in cloud applications, coordination is typically governed by state-dependent synchronization (e.g., mutual exclusion, causal chains) rather than dynamic priority rules. Moreover, the JavaBIP runtime [13] does not support user-defined priorities. Instead, it applies a

[1] `https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/IFinder/-/tree/real-time-marius`

default maximal-progress policy, in which larger interactions (in terms of the number of participating ports) are prioritized, sufficient for most practical scenarios. Thus, our approach targets the BI(P) fragment, where Priority is either absent or implicitly defined, ensuring compatibility with JavaBIP while retaining sufficient expressiveness for cloud coordination.

3.1.1. JavaBIP: exogenous coordination in Java

JavaBIP is a Java implementation of the BIP framework, enabling the exogenous coordination of concurrent components. Component behaviors are represented as FSMs using Java annotations on classes, methods, and parameters. The interaction model is specified via glue code in a *GlueBuilder* class, which defines JavaBIP macros to enforce synchronization constraints. Two key macros are:

- *requires*: Specifies that a port can participate in an interaction only if certain other ports also participate. For a port p, the constraint $p \Rightarrow a_1 \vee a_2 \vee \cdots \vee a_n$, where p is expressed as

```
port(Peer.class, "speak").requires(Tracker.class, "broadcast");
```

- *accept*: Ensures that if a port participates, it must be accepted by all other participating ports, formally $p \Rightarrow \bigwedge_{\substack{q \in P \setminus a \\ q \neq p}} \overline{q}$.

Data transfer between components is declared using $data(...).to(...)$, enabling state-dependent coordination (e.g., passing registration status from Tracker to Peer).

3.1.2. iFinder: compositional verification

iFinder is a verification tool for BIP systems that employs compositional verification to combat state-space explosion. Instead of analyzing the entire system monolithically, iFinder verifies components in isolation, using interaction invariants, logical constraints that define permissible synchronization patterns.

The system model, invariants, and negated safety properties are encoded into an SMT (Satisfiability Modulo Theories) problem. The SMT solver returns SAT (satisfiable) if a counterexample exists, so the system may violate the property. Otherwise, UNSAT (unsatisfiable) means the property holds and the system is correct with respect to the model. This approach enables scalable verification of deadlock-freedom and other safety properties.

3.2. Boolean formula processing

The translation from NaturalBIP to BIP/JavaBIP relies on Boolean reasoning and normal form transformations. We use PBL (Python Boolean Library)[2], a parser and manipulator for Boolean formulas based on Yacc [14], to handle this process. PBL supports conversion of formulas into Negation Normal Form (NNF) and Conjunctive Normal Form (CNF). In our compiler, we extend PBL to compute dual-Horn clauses, which characterize BIP connectors. A dual-Horn clause is a disjunction of literals with at most one negative literal

$$\neg p \vee a_1 \vee \cdots \vee a_n. \tag{1}$$

This corresponds to the causal rule $p \Rightarrow a_1 \vee \cdots \vee a_n$, meaning that for port p to interact, all a_i must participate. The set of dual-Horn clauses generated from a NaturalBIP specification

[2] `https://github.com/tyler-utah/PBL`

defines the allowed interactions in the system. This set is used to generate JavaBIP macros (via *requires* and *accepts*), construct BIP connectors in the Algebra of Connectors (AC), and ensure semantic equivalence between specification and implementation.

4. NATURALBIP

This section presents NaturalBIP, a DSL for specifying behavioral constraints in concurrent systems, and its compiler, which translates high-level specifications into formal BIP/-JavaBIP artifacts. The design of NaturalBIP is grounded in an ontology-driven architecture that reduces ambiguity in natural language while preserving expressiveness. The compiler automates the generation of verifiable models and executable coordination code, bridging the gap between informal requirements and formal implementation. We begin with an overview of the NaturalBIP methodology, followed by an explanation of the language's formal structure and the compiler's translation pipeline.

4.1. NaturalBIP language

We defined a grammar to derive pseudo-natural-language templates with placeholders for capturing requirements. Each derived pattern is associated with a formal representation in a logical language. Using these patterns addresses the ambiguity of natural language requirements. A boilerplate comprises attributes and fixed syntax elements. For example, in the boilerplate (2), "executes" is a fixed syntax element, while $\langle instance \rangle$ and $\langle action \rangle$ are placeholder attributes for user input.

$$\langle instance \rangle \quad \text{executes} \quad \langle action \rangle. \tag{2}$$

We propose an ontology architecture for specifying functional requirements. Based on the proposed ontology architecture, we define the syntax for our language in the context-free grammar form to interpret functional requirements and parse them to generate BIP connectors.

Behavioral Ontology (BO) contains concepts that specify functional requirements and their relationships, including components, actions, and states. These concepts are semantically interrelated, as shown in the conceptual overview in Fig. 1.

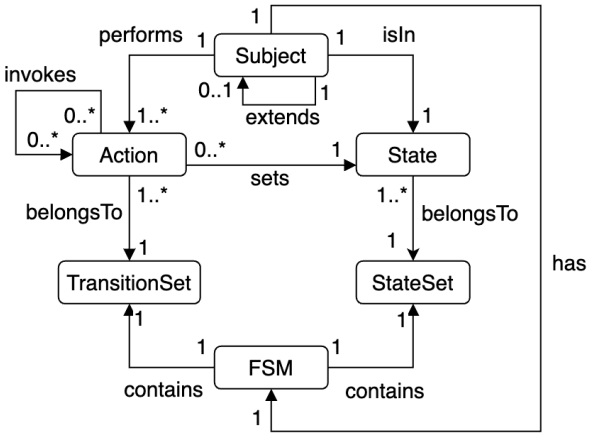

Figure 1: Behavioral Ontology (BO)

Fig. 2 presents the ontology framework designed to encode the structural and semantic elements of requirement specifications in the NaturalBIP language. This ontological model establishes a formal foundation for representing complex functional requirements through a hierarchical decomposition of interconnected components.

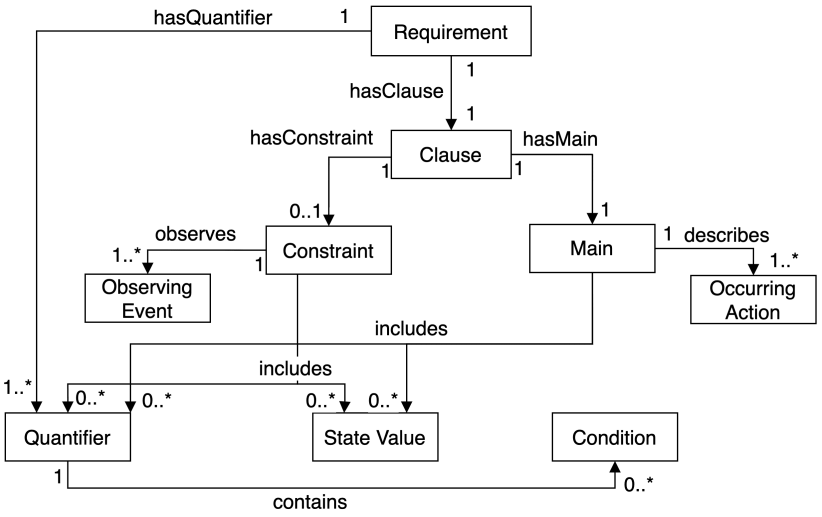

Figure 2: Conceptual model of the NaturalBIP Requirement Ontology (NRO)

The primary ontological elements, including *Quantifier*, *Condition*, *Occurring Action*, *State Value*, and *Observing Event*, are implemented as boilerplate syntactic constructs. These constructs incorporate both fixed structural elements and configurable placeholders, facilitating the systematic specification of requirements while maintaining consistency across different types of functional requirements. This approach ensures that requirements specifications adhere to standardized patterns while preserving the flexibility needed to express diverse system behaviors and constraints.

The syntax of NaturalBIP is defined by a context-free grammar that is left-associative, allowing requirements to be expressed in a structured, pseudo-natural language. A *Requirement* is formally structured as

Requirement ::= (Quantifier)+ [Constraint ","] Main

The core components are defined as follows:

- Quantification: Defines the scope of component instances using patterns such as *for all*, *for any*, or *there is a*.
- *Main* & Compound Expressions: The *Main* clause specifies the primary coordination logic. It consists of *Occurring Actions* (e.g., *p shall listen*, *p shall either speak or listen*) or *State Values* (e.g., *p is in Registered state*) connected by logical operators such as *and*, *or*, or *synchronized with*.
- *Constraints*: These are optional functional prerequisites following conjunctions like *if*, *while*, *after*, or *before*. A *Constraint* typically contains an *Observing Event* (e.g., *t executes broadcast*) or specific *State Values*.
- Conditions: Logical expressions that describe prerequisites using the template: *instance is/has/can phenomenon [instances]*.

To handle temporal and conditional constraints, NaturalBIP supports several derivable clause structures. The semantics of these clauses are defined as follows:

- *If (Constraint), (Main)* Specifies a global causal dependency where *Main* occurs immediately following the occurrence of *Constraint*.
- *While (Constraint), (Main)* Indicates that the *Main* action occurs globally during the period when the *Constraint* is observed to be true.
- *After (Constraint), (Main)/(Main) after (Constraint)* Defines an eventual temporal relationship where the *Main* clause occurs at some point after the occurrence of the *Constraint*.
- *Before (Constraint), (Main)/(Main) before (Constraint)* Specifies that the *Main* clause can be observed up until the point when the *Constraint* occurs.

Observing event, *Occurring Action*, *State Value*, *Condition*, and *Quantifier* are atomic components that contain placeholders to be filled with information adhering to the design of the target application (i.e. DSO). *Quantifier*, *Condition*, *Occurring Action*, *State Value*, and *Observing Event* are boilerplate syntax containing fixed elements and placeholders.

State value. This element describes that an instance of a *Subject* reaches a specific state. For example, "*p is in Registered state*" expresses that instance p is in the state *Registered*.

Observing event. This element is used in *Constraint* to specify the observation of an event. For example, "*t executes broadcast*" denotes the observation of an event that *Tracker* t executes the *broadcast* action, where t is an instance variable of *Tracker*.

Occurring action. This component specifies the occurrence of actions in *Main* including the execution (i.e. **instance** *shall* **action**, **instance** *shall not* **action**), and exclusive disjunction operator (i.e. "(*shall either ... or ...*)").

- "*p shall listen*" expresses that action *listen* of *Peer p* must be executed.
- "*p shall not speak*" expresses that action *speak* of *Peer p* must not be executed.
- "*(p shall either speak or listen)*" expresses that two actions *speak* and *listen* of instance p cannot occur at the same time.

Conditions. This element describes the constraints or conditions to execute actions. The template of this block is defined in Formula (3)

instance *is/has/can* phenomenon [**instances**]. (3)

Following the template, the first placeholder **instance** is the affected object of other **instances** if they exist. The phenomenon is defined by the collocation of lexical units of English from the Gellish English Dictionary [15] and the WordNet Lexical Database [16].

4.2. NaturalBIP compiler

In Section 4.1, we have proposed the NaturalBIP language for writing functional specifications. In this section, we propose an automated generation of artifacts, including the JavaBIP macro, data transfers, and constraints, from functional requirements written in the NaturalBIP language. Fig. 3 shows the overall process of the NaturalBIP compiler step by step, together with the corresponding input and output data:
Input: (i) the application design, and (ii) the NaturalBIP requirements
Output: JavaBIP artifacts for the implementation

- Step 1. *Pre-processing*: Analyzing the NaturalBIP requirements to obtain information on Quantifier, Interaction, and Condition, and skolemizing the requirements.

- Step 2. *Boolean Encoding*: Generating Boolean formulas from requirements by handling patterns.
- Step 3. *Conjunctive Normal Form (CNF) Transformation*: Using PBL, an open-source tool for handling Boolean formulas following algorithms proposed in [17], to generate CNF clauses.
- Step 4. *Dual-Horn Generator*: Calculating dual-Horn clauses from the CNF obtained in the previous step. The dual-Horn clauses present the interactions between the system's components [6].
- Step 5. *BIP Generator*: Generating artifacts, including JavaBIP macros, data transfers, conditions, and BIP connector in the form of the algebra of connectors (AC) [6].

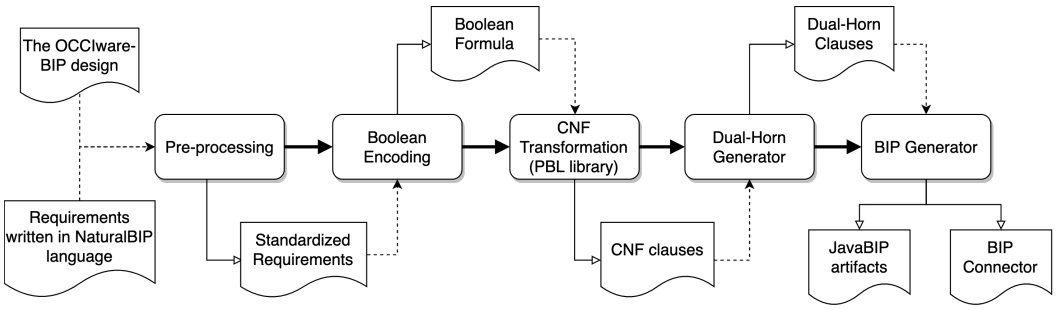

Figure 3: The process for generating JavaBIP artifacts from NaturalBIP requirements

4.2.1. JavaBIP artifacts generation

JavaBIP macros represent BIP connectors in Java implementation to coordinate the components' activities at runtime. Algorithm 1 presents how to generate the macro codes from the dual-Horn clauses representing the systems of causal rules. For each clause in the considered dual-Horn set (lines 5 and 6 in Algorithm 1), if it is a non-constant clause (lines 8), save it into the `requires` set (line 9) to generate corresponding JavaBIP macro for "requires" (lines 17-19). For example, the clause "∼(p_speak) | TP02speakptt_broadcast" is used to synthesize JavaBIP macro "port(Peer.class, "speak").requires(Tracker.class, "broadcast")". This transformation replaces the instances' names with the corresponding class names in the design (line 16).

Each sub-set in the `dualHorn_clauses_sets` contains a clause presenting the causal rule "$true \Longrightarrow \vee_{i=1}^{n} p_i$" (line 10). The set of `accepts` is computed by the union operation of such clauses (line 11) in all the sub-set. Based on the `accepts`, we generate "`accepts_macro`" by each element in `accepts` accepts all the others (lines 20-22).

The `GlueBuilder` contains JavaBIP macros and data transfers between components at runtime. To generate such information, we consider the conditions. If the condition template is "$\langle instance_1 \rangle$ **is/has/can something** $\langle instance_i \rangle$" (where $i \geq 0$) and the subject of $\langle instance_1 \rangle$ is different from the subject of $\langle instance_i \rangle$, there is a data transfer from $\langle subject_i \rangle$ to $\langle subject_1 \rangle$. Thus, the generated data transfer for requirement **TP_02** is "data(Tracker.class ,"Tracker2Peer_data").to(Peer.class ,"Tracker2Peer_data");". This data transfer is generated from condition "`is_registered_to`".

Algorithm 1 JavaBIP macro generation

```
1: Input: dualHorn_clauses
2: Output: javaBIP_GlueBuilder
3: init accepts
4: init requires
5: for sub_set ∈ dualHorn_clauses_sets do
6:     for clause ∈ sub_set do
7:         if clause.pos ≠ ∅ then
8:             if clause.neg ≠ ∅ then
9:                 requires := requires ∪ {clause}
10:            else
11:                accepts := accepts ∪ clause.pos
12:            end if
13:        end if
14:    end for
15: end for
16: //replace instance names with corresponding class names in requires and accepts
17: for clause ∈ requires do
18:    require_macro += clause.neg requires ",".join(clause.pos)
19: end for
20: for elm_i ∈ accepts do
21:    accepts_macro += elm_i accepts ",".join (accepts \ {elm_j})
22: end for
23: return require_macro + accept_macro
```

JavaBIP classes contain `@Guard` functions that specify the condition to fire the FSM's transitions. Each condition is considered a `@Guard` (i.e., a boolean function). The boolean functions are generated from a template, with the default return value set to "*True*". Depending on the requirements, developers write the actual content for those functions. The parameters of those functions are generated depending on the kind of $\langle subject_i \rangle$:

- If $\langle subject_i \rangle$ is empty, there is no parameter,
- If $\langle subject_i \rangle$ is the same as $\langle subject_1 \rangle$, generate a normal parameter,

```
In Peer.java:
@Guard(name="is_different_from")
public boolean is_different_from (Peer Peer_ins) { return true;}
```

- If $\langle subject_i \rangle$ is different from $\langle subject_1 \rangle$, the generated parameter is a `@Data` received through the specified data transfers.

```
In Peer.java:
@Guard(name="is_registered_to")
public boolean is_registered_to (@Data(name="Tracker2Peer_data") Tracker Tracker_ins) {
    return true;
}
```

4.2.2. BIP connectors generation

From a set of atomic interactions computed from the system of causal rules. Let $X = \{p \Longrightarrow x_p\}_{p \in P^t}$ be a saturated system of causal rules, with

$$x_p = \bigvee_{i=1}^{n_p} a_i^p,\ atomic_interaction_set = \{pa_p^p | p \in P \cup \{true\},\ i \in [1, n_p]\}.$$

The output is BIP connectors, which are the textual representation of $\mathcal{AC}(P)$. Listing 1 illustrates the generated BIP connectors (lines 7-9) from the sets of atomic interactions (lines 2-4) conforms to the saturated systems of causal rules.

Listing 1: Generated BIP connectors

```
1 // sets of atomic interactions
2 {{{'TP02listenptt_broadcast'}, {'p_listen', 'TP02listenptt_broadcast'}, {'p1_listen', '
      TP02listenptt_broadcast'}},
3 {{'p1_listen', 'TP02listenptt_broadcast'}, {'p_speak', 'TP02speakptt_broadcast'}, {'
      TP02listenptt_broadcast'}, {'TP02speakptt_broadcast'}},
4 {{'p1_listen', 'TP02listenptt_broadcast'}, {'TP02listenptt_broadcast'}}}
5
6 // corresponding generated BIP connectors
7 (Tracker.broadcast)'-(Peer.listen)-(Peer.listen)
8 (Tracker.broadcast)'-(Tracker.broadcast)'-(Peer.speak)-(Peer.listen)
9 (Tracker.broadcast)'-(Peer.listen)
```

5. OCCIWAREBIP FRAMEWORK WITH NATURALBIP

This section presents the integration of NaturalBIP into OCCIwareBIP [5], a model-driven framework for designing, verifying, and implementing concurrent cloud applications. The integration demonstrates how NaturalBIP enhances the design phase by enabling early specification of coordination policies and automated generation of verifiable and executable artifacts.

The OCCIwareBIP toolchain extends the OCCIware metamodel to support exogenous coordination via BIP/JavaBIP. In this workflow, the **Specification** concept is introduced to embed NaturalBIP requirements directly into the system design (Fig. 4). This allows cloud architects to define behavioral constraints at the modeling level, rather than deferring them to the implementation phase.

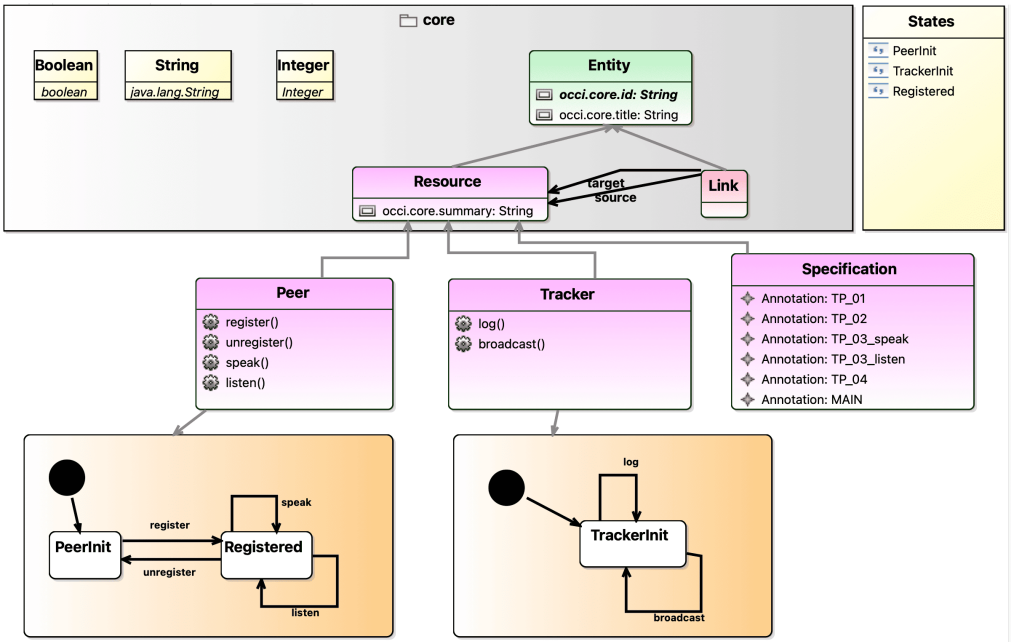

Figure 4: The OCCIware design of the Trackers-Peers example

5.1. Extending the OCCIware metamodel

The OCCIware design supports developers in specifying cloud components and their structural constraints. In contrast, developers impose behavioral constraints during the implementation phase. With this approach, they must track and analyze affected codes whenever behavior constraints change, a time-consuming process. To support NaturalBIP, we extend the OCCIware metamodel with two key concepts:

- **Specification**: A container for NaturalBIP requirements, associated with components or the entire system. It includes (i) a list of NaturalBIP-compliant requirements; and (ii) A `MAIN` annotation to select active requirements (e.g., *all*, $exceptTP_02_unregister$).

- **actiontype**: A new property for transitions in the OCCIware FSM, classifying actions as (i) Spontaneous: externally triggered (e.g., *register*); (ii) Enforceable: must be synchronized via coordination (e.g., *speak*); and (iii) Internal: state updates without external interaction.

Domain-specific ontology for the cloud domain. To express specification from the cloud domain, we construct a DSO encompassing the cloud domain by mapping the OCCIware metamodel concepts to the corresponding classes in the Behavioral Ontology as shown in Table 1. This mapping ensures semantic consistency between the cloud application model and NaturalBIP specifications.

Table 1: Mapping between the BO classes and the OCCIware metamodel concepts

OCCIware metamodel concepts	Behavioral Ontology classes
Kind	Subject
Mixin	Subject
FSM	FSM
Transition	Transition
State	State
Action	Action

5.2. Artifact generation for verification and implementation

The design can be verified by checking whether it satisfies the specified behavioral constraints and is free from deadlocks. In this work, we use iFinder, a compositional deadlock detection and verification tool for BIP models. iFinder expects a BIP model, a *.inv* file, and a set of linear safety properties on input. The *.inv* file specifies the instructions for calculating the system's invariants. It is generated as follows:

- Each component in the OCCIwareBIP design is generated following the template "−at <OCCIwareBIP Component> −a atom−control".
- Specifying a compound element to control the whole system "−ct <Application_Name>Compound −a control−reachability".

5.3. Integration of JavaBIP into OCCIware implementation

Following the exogenous approach, the application is implemented by separating the computation and coordination codes. The computation codes are Java classes generated by mapping information from the OCCIware design, where:

- Kind and Mixin components in the OCCIware design are Java classes annotated with `@ComponentType`.
- Each action in the OCCIware component is a function with annotation `@Transition` labeled by the action's name from one state to another state or itself.

The coordination code that describes the interaction between components and data exchange is the *GlueBuilder* class, which is generated as described in Section 4.2.1.

In [6], it is shown that *the connectors and the set of dual-Horn clauses generated from define the same set of interactions*. The textual representations we generate for JavaBIP (GlueBuilder macros) and BIP (connectors) correspond precisely to the dual-Horn clauses and the generated connectors. Therefore, *they are equivalent* among themselves. Thus, the verification results obtained for the BIP model remain valid for the JavaBIP model.

5.4. Experimental results

We have demonstrated the usage of our NaturalBIP language to specify the behavioral constraints of the Trackers-Peers example. By providing the NaturalBIP compiler, we save developers time and effort. Instead of learning to write BIP connectors (e.g., connector `c = (s.act)'−[(t.act)'−(v.act)]`), BIP models[3] for the verification and JavaBIP macros[4] for the implementation, our toolchain supports generating those artifacts automatically from the specifications written in NaturalBIP language.

Table 2: The number of lines for writing the constraints to describe the Trackers-Peers system

Functionality	NaturalBIP specification (number of specifications)	Generated artifacts (lines of code)		
		BIP connectors	JavaBIP macros	Connectors in the BIP model
3	**5**	3	13	*396*
		412		

Table 2 compares the number of lines needed to specify the functionality, which is choosing the third policy. Instead of writing 412 lines to specify BIP and JavaBIP artifacts, developers need only five lines to specify the requirement in our NaturalBIP language. Since the number of possible connectors in the BIP model *increases exponentially* with the number of components, manually specifying and managing these connectors becomes increasingly cumbersome and error-prone, hindering the scalability and usability of the BIP framework. For seven instances, including three Trackers and four Peers, developers must specify 396 connectors (each connector is written on a single line). Meanwhile, the number of NaturalBIP specifications is kept the same. Thus, saving time and effort in designing and verifying systems using BIP.

[3] `https://github.com/TrinhLK/NaturalBIP-Compiler/blob/master/test/gen-data/trackerpeer.bip`

[4] `https://github.com/TrinhLK/NaturalBIP-Compiler/blob/master/test/gen-data/JavaBIP_GlueBuilder.txt`

We also validate our proposal by creating Heroku Deployer [18], a microservice that automatically deploys a web application or microservice on the Heroku platform.

- *HerokuDeployer* is a microservice designed using an exogenous strategy to automate the deployment of a Web application named the Monitor-Switch on Heroku. It operates on Infrastructure as a Service (IaaS) platforms, such as Google Compute, or on a local machine by interacting with Heroku APIs and executing Heroku CLI commands via the terminal.

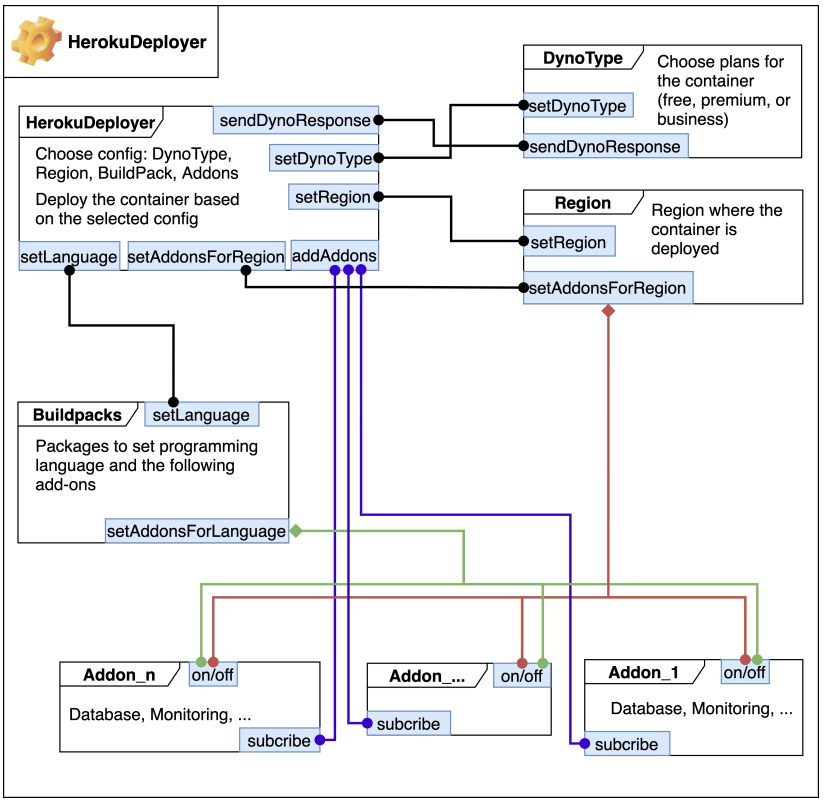

Figure 5: The overview of HerokuDeployer microservice

Fig. 5 provides an overview of the HerokuDeployer microservice. The type of the deployed *Heroku Dyno* is determined through two connectors linking *HerokuDeployer* to the DynoType components. Likewise, the region, addons, and programming language of the *Dyno* are defined via connectors from *HerokuDeployer* to the *Region* and *Buildpacks* components. Since each programming language supports only a specific set of addons, a diamond arrow (representing a trigger connector in BIP) connects the *setAddonsForLanguage* function in *Buildpacks* to the *Addons* component. Similarly, another diamond arrow links the *setAddonsForRegion* function in *Region* to the *Addons* component.

- *Monitor-Switch* is a web application that demonstrates the benefits of the exogenous approach, particularly in maintaining modularity. It includes multiple policies for responding to different events. With this approach, users can define these policies as separate components and select them based on the specific synchronization constraints required.

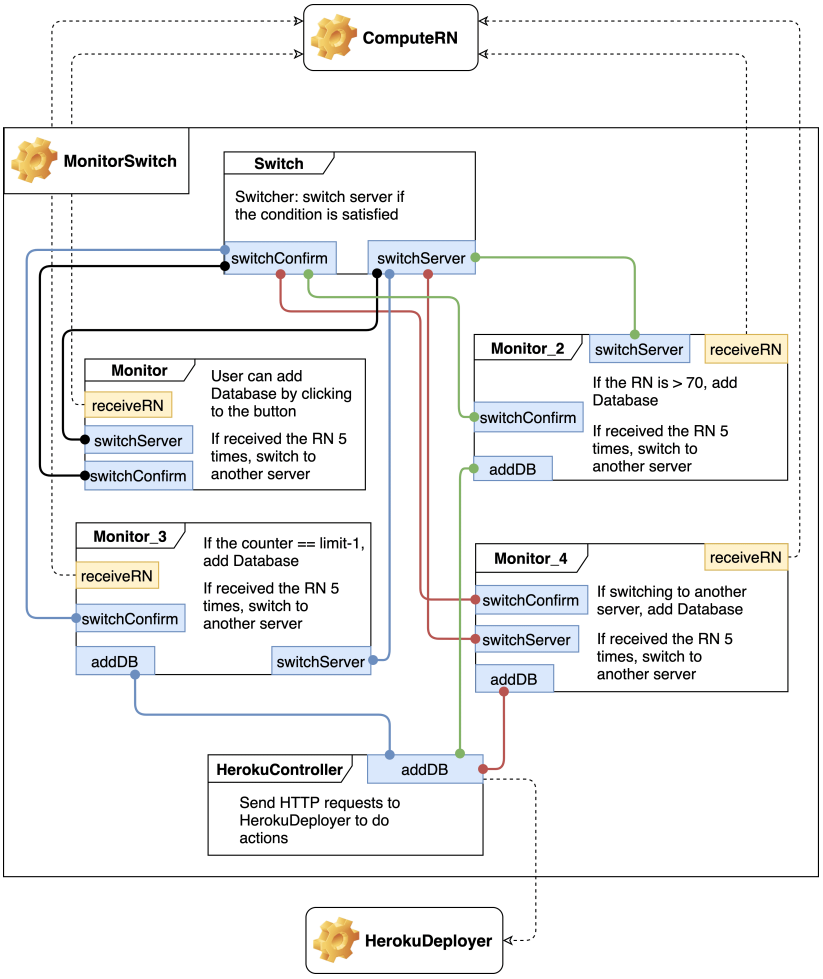

Figure 6: The overview of the Monitor-Switch Web application

Fig. 6 provides an overview of the *Monitor-Switch* web application. It includes four `Monitor` components (`Monitor`, `Monitor_2`, `Monitor_3`, and `Monitor_4`) that add a database add-on to the existing `Heroku Dyno`. The `HerokuController` component serves as an API that calls a function from `HerokuDeployer` to create the database addon.

- *Compute* is a lightweight microservice designed to generate a random integer.

Row 1 of Table 3 compares the lines of code required to define the functionality of selecting Java as the deployment language for a Heroku Dyno. Using BIP connectors, writing verification code in the BIP model, and implementing JavaBIP macros would require approximately 100 lines of code. In contrast, the same requirement can be specified in just 7 lines using our NaturalBIP language. Similarly, Row 2 of Table 3 highlights that specifying all deployment functionalities with NaturalBIP requires only 105 lines, whereas using BIP connectors (105 lines), writing verification code in the BIP model (496 lines)[5], and implementing JavaBIP macros (774 lines[6]) results in a total of 1375 lines.

[5] `https://github.com/TrinhLK/GeneratingBIPFile/blob/master/output/herokudeployer.bip`
[6] `https://github.com/TrinhLK/JavaBIPonCloudDemo/blob/main/HerokuDemo/src/main/java/`

Table 3: The number of lines for writing the constraints to describe the functionality, choose Java language and the whole functionalities in deploying a Web application onto a free Heroku Dyno.

Functionality	NaturalBIP specification (number of specifications)	Generated artifacts (lines of code)		
		BIP connectors	JavaBIP macros	Connectors in the BIP model
Choose Java Language	**7**	7	50	*43*
		100		
All	**105**	105	774	*496*
		1375		

6. CONCLUSION

This paper presented NaturalBIP, an ontology-driven domain-specific language for specifying functional requirements of exogenously coordinated concurrent software systems. By combining controlled grammar rules, specification templates, and ontology-based semantics, NaturalBIP helps reduce ambiguity in natural-language requirements while making semantic relationships and implicit assumptions explicit through the NaturalBIP Reference Ontology (NRO). We also showed how domain-specific ontologies can be developed and integrated to support requirements specification in specific application domains.

We further developed a NaturalBIP compiler that translates NaturalBIP specifications into three main artifacts: BIP models for formal verification with iFinder, Java classes annotated with JavaBIP to represent component behaviors, and JavaBIP GlueBuilder code to implement synchronization constraints. The compiler also generates Boolean guard conditions and detects basic specification errors, such as undefined actions or states.

The feasibility and effectiveness of the proposed approach were demonstrated through its application to the OCCIwareBIP framework and the Trackers-Peers case study. The results indicate that NaturalBIP can reduce the effort required to specify, verify, and implement systems that follow the exogenous coordination approach, primarily by enabling developers to modify coordination requirements at the specification level rather than directly modifying implementation artifacts.

Several directions remain for future work. First, NaturalBIP should be extended to support richer behavioral specifications, including temporal constraints and hierarchical states inspired by Statecharts and SysML. Second, rigorous empirical studies with software developers are needed to evaluate its impact on productivity, usability, and error reduction beyond line-count-based assessment. Third, the completeness and quality of the cloud-domain ontology should be further validated by analyzing inheritance structures, redundancy, inconsistency, and coverage of cloud concepts. In addition, the applicability of NaturalBIP should be explored beyond OCCIware Studio, particularly in other FSM-based design frameworks

`occideployer/GlueBuilder_Specification.java`

and domains such as IoT, embedded systems, and microservices orchestration.

Finally, the tooling ecosystem should be improved through an online IDE with syntax highlighting, real-time error checking, live compilation feedback, and integration with CI/CD pipelines for automated verification and deployment. Scalability benchmarking on large-scale synthetic models is also necessary to assess the performance of the NaturalBIP compiler and its suitability for industrial-grade concurrent and distributed systems.

REFERENCES

[1] M. Hanikat, "Towards a correct-by-construction design flow: A case-study from railway signaling systems," Master's Thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science (EECS), Stockholm, Sweden, 2021.

[2] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '06, USA: IEEE Computer Society, 2006, pp. 3–12, ISBN: 0769526780. DOI: 10.1109/SEFM.2006.27.

[3] F. A. Zaraket, M. Jaber, M. Noureddine, and Y. Falcone, "From high-level modeling toward efficient and trustworthy circuits," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 2, pp. 143–163, 2019.

[4] M. Bernardo and F. Franzè, "Exogenous and endogenous extensions of architectural types," in *Coordination Models and Languages*, F. Arbab and C. Talcott, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 40–55, ISBN: 978-3-540-46000-8.

[5] T. Le-Khanh, H.-G. Nguyen, S. Bliudze, and P. Merle, "Towards exogenous coordination of concurrent cloud applications," *International Journal of Software Engineering and Knowledge Engineering*, vol. 34, no. 01, pp. 1–25, 2024. DOI: 10.1142/S0218194023500389.

[6] S. Bliudze and J. Sifakis, "Causal semantics for the Algebra of Connectors (extended abstract)," in *FMCO 2007*, F. de Boer and M. Bonsangue, Eds., ser. LNCS, See the journal version, Berlin Heidelberg: Springer-Verlag, 2008, pp. 179–199. DOI: 10.1007/978-3-540-92188-2_8.

[7] N. Mahmud, C. Seceleanu, and O. Ljungkrantz, "Resa: An ontology-based requirement specification language tailored to automotive systems," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, IEEE, 2015, pp. 1–10.

[8] T. H. Nguyen, B. Q. Vo, M. Lumpe, and J. Grundy, "Kbre: A framework for knowledge-based requirements engineering," *Software Quality Journal*, vol. 22, no. 1, pp. 87–119, 2014.

[9] M. Bozga, M. Jaber, N. Maris, and J. Sifakis, "Modeling dynamic architectures using Dy-BIP," in *Software Composition*, T. Gschwind, F. De Paoli, V. Gruhn, and M. Book, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–16, ISBN: 978-3-642-30564-1.

[10] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical. Structures in Comp. Sci.*, vol. 14, no. 3, pp. 329–366, Jun. 2004, ISSN: 0960-1295. DOI: 10.1017/S0960129504004153.

[11] F. Gadducci and U. Montanari, "The tile model," in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. P. Stirling, and M. Tofte, Eds., The MIT Press, May 2000, ISBN: 9780262281676. DOI: 10.7551/mitpress/5641.003.0010.

[12] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel, "Compositional semantics for open petri nets based on deterministic processes," *Mathematical Structures in Computer Science*, vol. 15, no. 1, pp. 1–35, 2005.

[13] A. Mavridou, V. Rutz, and S. Bliudze, "Coordination of dynamic software components with JavaBIP," in *Proceedings of the 14th International Conference Formal Aspects of Component Software (FACS)*, ser. Lecture Notes in Computer Science, vol. 10487, Springer, 2017, pp. 39–57. DOI: 10.1007/978-3-319-68034-7_3.

[14] S. C. Johnson, "Yacc: Yet another compiler-compiler," AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Tech. Rep., PS1:15-1 - PS1:15–32.

[15] A. S. H. P. Renssen, *Gellish: A Generic Extensible Ontologial Language: Design and Application of a Universal Data Structure*. Amsterdam: DUP Science, 2005.

[16] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995, ISSN: 0001-0782. DOI: 10.1145/219717.219748.

[17] A. R. Bradley and Z. Manna, *The calculus of computation: decision procedures with applications to verification*. Berlin, Heidelberg: Springer-Verlag, 2007, ISBN: 3540741127.

[18] T. Le-Khanh, "Design of correct-by-construction self-adaptive cloud applications using formal methods," Theses, Université de Lille, Jan. 2023. [Online]. Available: https://theses.hal.science/tel-04136838.

Received on May 07, 2025
Accepted on December 03, 2025