Towards a Domain-specific Language for Developing Concurrent Applications using BIP Framework

Trinh Le-Khanh (✉ trinhlk@vnu.edu.vn)  
University of Engineering and Technology, VNUH

Son Nguyen  
University of Engineering and Technology, VNUH

Research Article

Keywords: Behavior-Interaction-Priority, Concurrent System, Correct-by-Construction, Domain-specific Language, Exogenous Coordination

Posted Date: April 13th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-2796017/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.
Towards a Domain-specific Language for Developing Concurrent Applications using BIP Framework

Trinh Le-Khanh and Son Nguyen
University of Engineering and Technology, Vietnam National University, Hanoi, Vietnam.

Contributing authors: trinhlk@vnu.edu.vn; sonnguyen@vnu.edu.vn;

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 designing a system, designers first analyze the system’s requirements to construct BIP connectors describing the coordination between the system’s components. Based on these connectors, developers create a BIP model for verification that uses model checkers and JavaBIP artifacts for implementation following an exogenous approach. However, constructing the BIP connectors and creating BIP/JavaBIP artifacts require a certain level of 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 introducing 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

Modern software systems are intrinsically concurrent, where any software entity that goes beyond simply computing a function has to interact and share resources with other entities. Correctly coordinating concurrent entities’ access to these resources is crucial to meeting user and system requirements and preventing deadlock situations and operational issues (e.g., race conditions or synchronization overhead). In traditional software development, the code coordinating entities’ access to available resources is interleaved with the computation code of software components (e.g., locks, semaphores, and monitors), complicating application maintenance when facing policy changes. Although change impact analysis can aid in maintenance, this process can be time-consuming and labor-intensive. Exogenous models and languages (e.g., Tile model [4], Reo [5], and BIP [6]) have been introduced to address this problem [7]. The exogenous approach separates computation and coordination code, increasing components’ reusability and benefiting software’s modularity.

In a system, the failures found at the late phase of the development process are common and might cause severe consequences during deployment. The late repair of these errors or rebuilding of the system can be expensive and time-consuming. Therefore, early fault detection of system requirements and fixing them is essential in software development. One of the popular methods to proactively prevent potential errors is applying the correct-by-construction (CbyC) approach, which combines formal methods and incremental developments to verify the system’s requirements [9]. In CbyC, developers create a formal model from the requirements. The formal model allows developers to prove the requirements’ consistency and correctness. This early fault detection process by CbyC could greatly reduce the consequence caused by potential faults in the implementation phase.

The Behavior-Interaction-Priority (BIP) framework supports the exogenous coordination and CbyC approach for building complex designs from smaller designs to enforce given properties. However, using this framework requires a certain expertise in formal methods, model-driven design, and concurrent systems, and they need to conduct many manipulations. Thus, developers must invest time and resources in learning the framework and the associated tools and techniques. Consider a requirement describing that “the system shall ensure that actions \(p, q, \) and \(r\) of components \(C_1, C_2,\) and \(C_3\) can run simultaneously without causing conflicts or errors. The system should be designed to coordinate the execution of these actions so that they do not interfere with each other and can operate independently. Additionally, the system should provide mechanisms for monitoring the execution of each action and detecting any potential issues or errors that may arise during their simultaneous execution”. After analyzing the requirement, developers design the corresponding BIP connector, which is the synchronization between three ports as shown in Fig. 1. From the designed BIP connector, developers create (1) a BIP model for verifying whether the designed system is deadlock-free; and (2) JavaBIP artifacts for implementation, including Java classes with JavaBIP
natural annotations and JavaBIP Gluebuilder—an external file specifying the coordination between components. For example, learning to understand and correctly use the causal and acceptance constraints (i.e., requires and accepts codes in Fig. 1) in JavaBIP glue specification takes time and effort. A small difference in this specification leads to another coordination policy. This analysis of the requirements and creation of BIP/JavaBIP artifacts requires expertise in formal methods, model-driven design, and concurrent systems. As a result, developers need to invest time and effort to learn and use the BIP framework and the associated tools properly.

Fig. 1: BIP and JavaBIP artifacts.

In this work, our main objective is to provide software designers with means to specify behavioral constraints naturally to reduce manual effort and time to learn using the BIP. These specifications are the input to validate the system requirements and design by applying correct-by-construction techniques. To this end, we address the following research questions:

- **RQ1**: How do we provide designers with means to write behavioral constraints in a language that is easy to learn and use?
- **RQ2**: How can these specifications be used to verify and validate the target system?

Fig. 2 provides an overview of our contributions. We address RQ1 by proposing NaturalBIP, a pseudo-natural language for specifying functional requirements. This language supports designers in writing unambiguous specifications of system functionalities. To address the RQ2, we provide NaturalBIP Compiler, a tool taking the specification written in NaturalBIP language as the input to generate: (1) a formal model is used as the input for an appropriate tool to ensure that the target system operates safely, and; (2) Java classes with empty functions and JavaBIP annotations. In particular, each generated Java class represents a component in the OCClware design. Then, developers write those functions to complete the component’s functionalities. To this end, our proposed language and tool allow specifying behavioral constraints in a language...
similar to English sentences and transforming them into necessary manipulations for developing correct-by-construction concurrent applications using the BIP framework.

The rest of the article is structured as follows. The next section discusses the related work. Section 3 introduces the background knowledge necessary for this paper and some frameworks and tools our proposed framework is built upon. Section 4 presents our proposed language. 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. An implementation of the proposed language is described in Section 5. Finally, Section 6 concludes the paper and discusses future directions.

2 Related work

The system design aims to define the system’s architecture, components, or data to satisfy requirements [10]. Validating the design is needed to ensure the specification is valid for the implementation. Natural languages are ambiguous [11]; hence, they are unsuitable for the specification of requirements to create a quality design. Formal model-based development is the process of implementing the system correctly. In this process, requirements are transformed into formal specifications [12], which describe the system’s structure and behaviors together with external stimuli [13]. Then, the formal specification will verify some properties, such as consistency or completeness.
2.1 Requirement Engineering

An ontology-driven specification is an approach for formulating system requirements in natural language. Numerous research on requirement engineering based on ontology [14], including representing domain knowledge [15–19], formalizing requirements [20, 21], verifying some properties of the requirements (e.g., consistency [22–24], completeness [25–27]), or validating requirements [20, 22, 28]. In particular, starting from an upper ontology describing the general concepts for use in multiple contexts, developers define the domain-specific ontology to define domain-specific concepts and their relationship.

Pattern-based specifications are textual templates filled by defined ontology concepts. Restricting the specification to terms from an ontology removes the ambiguity of the natural language. To ensure a unique interpretation for each specification, some approaches employ requirement boilerplates [28–30]. However, the more boilerplates, the more effort and time developers need to learn and apply them correctly. In our work, we provide templates, which are structures with placeholders that ontology concepts can replace. By defining the semantics of each template, we ensure the unique interpretation of each specification and flexibility in writing them. Therefore, reducing the effort and time to learning our language.

Ontology-Driven Conceptual Modeling (ODCM) [31] is the application of ontological theories to improve the theory and practice of conceptual modeling. Conceptual modeling represents aspects of the physical world using models for communicating, learning, and problem-solving between users. The ability to detect and correct errors depends on the conceptual models’ quality. However, many conceptual models lacked an adequate specification of terminologies and semantics of the underlying models, which led to the inconsistency of knowledge’s interpretation and use [32]. Therefore, ontologies were introduced to provide a foundation for conceptual modeling. Ontology is “the set of things whose existence is acknowledged by a particular theory or system of thought” [33]. Ontology can be used to evaluate conceptual modeling languages of frameworks (e.g., Web Ontology Language (OWL), Unified Modeling Language (UML)) [34], become the theoretical foundations of a conceptual model [35, 36], or improve semantic interoperability [37].

2.2 Component-Based Approach

Most programs are implemented using endogenous [7] coordination, where coordination codes are incorporated within computational codes of components. For complex distributed systems, it is hard to debug and maintain because developers must recheck all related components codes and other components interacting with them. Meanwhile, in the exogenous coordination [7] approach, each component implements its functions by only computation codes and interacts with the others through external ports. This high level of abstraction makes it easy to reuse components to build large-scale systems and tackle
the difficulty in debugging and maintaining the development following the endogenous coordination approach.

The computation and coordination are separated in the component-based design, where the computational entities are the system’s components. Connectors coordinate the interaction among these components. Some formal approaches for modeling, compositing, and analyzing connectors include the algebra of stateless connectors [38], the Tile model [4], nets with boundaries [39], process calculi, REO [5], and BIP [6].

- **The algebra of stateless connectors.** Bruni et al. [38] present an algebra of five basis stateless connectors. Then they used graphs to describe concrete structures, tiles to represent operational and observational semantics, and tick-tables to provide denotational semantics.

- **The Tile model.** The Tile model [4] provides a flexible and appropriate semantic configuration for concurrent systems [40, 41]. This configuration defines operational and abstract semantics to suitable connectors classes in the algebra of stateless connectors.

- **Process calculi.** Calculi such as CSP [42], CCS [43], the π-calculus [44], process algebras [45–47], and the actor model [48] are computation models to handle the complexities in constructing concurrent systems. Capizzi et al. [49] introduced the aspect-oriented technique to redesign a distributed program by altering only the coordinating aspects while keeping the computational code untouched. Scholten et al. [50] proposed a variant of the π-calculus that allows different processes to communicate and impose exogenous coordination through user-defined channel types.

- **The Reo coordination model.** Reo Scripting Language (RSL) [5] is a declarative channel-based language supporting exogenous coordination. This language is used with Constraint Automata Reactive Module Language (CARML) in Vereofy toolkit [51] to model the system. Vereofy also supports model checking of specific components and composite systems with safety properties written in Linear Time Logic (LTL) [52] and Branching-time Stream Logic (BTSL) [53].

- **The BIP framework.** Behavior-Interaction-Priority (BIP) [54, 55] is a framework providing a mechanism for coordinating concurrent components. Correctness-by-construction is a significant feature of BIP. Its composability can be used to preserve the deadlock-freedom of the underlying behaviors using appropriate tools such as iFinder [56]. JavaBIP [57, 58] is an open-source Java version of the BIP framework that relies on Java annotations, component APIs, and specification files to coordinate concurrent components.

Our work uses BIP and JavaBIP frameworks to support designing, verifying, and implementing correct-by-construction concurrent software.

### 2.3 Correct-by-Construction Software Development

System errors found at the late phase of the development process are popular [8]. Undetected system errors in the requirement analysis cause enormous
consequences in the deployment. Fixing these errors or rebuilding the system is costly and time-consuming. The Correctness by Construction (CbyC) approach is a solution to address this problem. The CbyC combines formal methods and incremental developments [9]. It starts by constructing a formal model from the system requirements. This model can be used as the input for model checkers or theorem provers to validate and verify whether the design meets its specification and whether the specification meets the requirements. This process has been applied in industry and demonstrated its effectiveness for decreasing errors and increasing productivity in developing safety-critical applications [59, 60] or distributed systems (e.g., cloud applications) [61].

In the implementation phase, developers choose the appropriate language to implement the design depending on the target system. The selected language must have supporting frameworks/tools to analyze and effectively verify correctness. For example, BIP and JavaBIP frameworks are suitable for developing concurrent software due to their rigorous and unambiguous semantics. In particular, BIP allows the creation of an executable model to verify whether the system under design satisfies a specific property (e.g., the deadlock-freedom). To ensure the high-level abstraction of the model, only relevant actions are represented as FSM’s transitions. All the allowed synchronizations (i.e., the interactions) between the concurrent components are specified in BIP glue.

3 Background Knowledge

This section introduces theoretical foundations and frameworks important to this work, including PBL - an open-source library to handle Boolean formulas, BIP and JavaBIP frameworks, and the iFinder - the model checker used to verify the deadlock-freedom of the target system.

3.1 Parser Generators

A parser generator tool generates a parser, which reads input text according to a set of rules (or grammar) and produces a structured representation of that text. By automating the process of creating parsers instead of writing a parser from scratch, parser generator tools save time and effort. Typically, the input to a parser generator tool is a formal grammar that specifies the syntax of a programming language or other structured format. The parser generator then produces source code for a parser in a programming language like C, Java, or Python. Parser generator tools can be used for various applications, such as creating compilers, interpreters, and other tools to process structured text. Some popular parser generator tools include Yacc [62], Bison, and ANTLR [63].

PBL is a simple Python Boolean library based on Yacc that parses and manipulates standard DIMACS (i.e., Discrete Mathematics and Theoretical Computer Science) files and a custom language. PBL provides functions to convert the Boolean formulas to the Negation Normal Form (NNE) or

1 https://www.gnu.org/software/bison/
2 https://github.com/tyler-utah/PBL
NaturalBIP Language

CNF. Listing 1 shows an example with a Boolean formula (line 2) and its corresponding CNF clauses (line 4).

Listing 1: A Boolean formula and its corresponding CNF clauses

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// Boolean formula</td>
</tr>
<tr>
<td>2</td>
<td>((x1 =&gt; y1 &amp; y2) &amp; (x2 =&gt; y2 &amp; y3)) &amp; ~(x3 =&gt; x4)</td>
</tr>
<tr>
<td>3</td>
<td>// CNF clauses</td>
</tr>
<tr>
<td>4</td>
<td>(~x1</td>
</tr>
</tbody>
</table>

Our work extends PBL to compute dual-Horn clauses for constructing coordination codes in BIP and JavaBIP (see Section 4.2).

3.2 The BI(P) Framework

Behavior-Interaction-Priority (BIP) [6] is a component-based framework for the design of correct-by-construction systems. By superimposing three layers: behavior, interaction, and priority, it provides a simple yet effective framework for managing concurrent components. In the simplified version, there is no Priority. Hence, we denote it as BI(P).

The first layer (Behavior) presents atomic components with fixed activities considered ports, which are pairwise distinct. The components are modeled as automata, in which sets of ports label their transitions.

**Definition 3.1 (Component).** A component \( C = (S, P, \to) \) is a transition system where \( S \) is the set of states, \( P \) is the set of ports, and \( \to \subseteq S \times P \times S \) is the set of transitions.

For all \( s, s' \in S \) and \( p \in P \), we write \( s \xrightarrow{p} s' \) to denote the labeled transition \((s, p, s') \in \to\). It means \( a \) is enabled in \( s \) (i.e., \( s \xrightarrow{p} \)) iff \( s' \) exists.

The second layer (Interaction) defines a mechanism to coordinate components. Interactions are sets of ports that determine the allowable synchronizations between components.

**Definition 3.2 (Interaction).** Let \( P \) be the set of ports, an interaction over \( P \) is a non-empty set \( a \), such that \( a \subseteq P \).

In [64], the authors defined \( \mathcal{AI}(P) \) to provide a clear and convenient notation for manipulating sets of interactions.

Priorities in the third layer of BIP (Priority) establish scheduling limits and resolve conflicts simultaneously when many interactions are enabled. We named Glue refers to the interaction and priority layers together. In brief, the BIP engine drives the execution of a BIP system by cyclically applying the following protocol:

1. At the current state, each component sends to the BIP engine all the possible transitions;
2. The BIP engine selects an interaction that meets the glue requirements, conducts the data transmission, and notifies all components involved;
3. The notified components perform the functions related to the corresponding transitions.

Connectors are used to define an interaction model in a structured manner. If all connected ports are synchrons, this is a rendezvous synchronization, i.e., a connector defines exactly one interaction comprising all its ports (Fig. 3b). If there are one or more trigger ports, this is a broadcast synchronization, i.e., the connector describes the interactions consisting of all non-empty subsets of the connected ports containing one or more trigger ports (Fig. 3c).

![Image of BIP connectors]

**Fig. 3**: BIP connectors with the corresponding sets of interactions.

In the hierarchical connector, the interaction from each subconnector forms an allowed interaction according to the port labeling of the connector nodes. For instance, the causal chain connector in Fig. 3d has a trigger port \( p \) and a synchron, which is a binary broadcast subconnector \( q \rightarrow r \). Thus the causal chain connector allows interactions are the single port \( p \) and any combination of \( p \) with some interaction of \( q \rightarrow r \). The subconnector \( q \rightarrow r \) allows interactions \( q \) and \( qr \). As a result, the allowed interactions of the causal chain are \( \{ p, pq, pqr \} \).

### 3.2.1 The JavaBIP Framework

JavaBIP [57] is a Java implementation of the BIP framework. Component behaviors in JavaBIP can be represented by a Finite State Machine (FSM), which has a finite number of states and a finite number of transitions between them. JavaBIP framework uses Java annotations associated with class, method, and parameter declaration to represent the FSMs.

In [65], the authors demonstrate that the interactions set defined by a BIP connector can be characterized by a Boolean formula that is a conjunction of
implications of the form:

\[ p \Rightarrow a_1 \lor \ldots \lor a_n \]  

(1)

with port \( p \) is considered as effect and each \( a_i \ (i \in [1, n]) \), being a conjunction of several ports, is considered causes. Obviously, for \( p \) to engage in an interaction, at least one \( a_i \) for \( i \in [1, n] \), in which all the belonging ports participate. Therefore, in an interaction, \( p \) can participate if there is some participation of \( a_i \). To specify the constraint (1) in JavaBIP, one uses the macro notation:

\[ p \text{ Require } a_1; \ldots; a_n \]

For example, the encodings of Broadcast 2 (Fig. 3c) is

\[ r \text{ Require } p; q \]

The macro \textit{Accept} defines that if a port \( p \) participates in an interaction, it must be accepted by all the participating ports in the considering interaction.

\[ p \text{ Accept } a, \text{ which formally means } p \Rightarrow \bigwedge_{q \in P \setminus a} q \neq p \]  

(2)

where \( P \) is the set of all ports of all the BIP components in the system.

In JavaBIP, the \textit{data transfer} is performed before the execution of the transition. At present, JavaBIP does not implement any \textit{priority} except the so-called \textit{maximal progress} priority applied by default to all JavaBIP systems, where interactions larger in terms of set inclusion are prioritized.

3.2.2 iFinder

iFinder [66] is the improvement of D-Finder [67], a model checker for verifying component-based systems described in the BIP language. iFinder proposed a compositional verification technique to avoid the combinatorial explosion of component-based systems. It uses interaction invariants for components instead of searching for adequate assumptions. The interaction invariants are considered “cooperation tests” [68] because they allow subtracting states infeasible by the semantics. iFinder consists of \textit{ifinder} and \textit{ichecker}\footnote{https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/IFinder/-/tree/real-time-marius}. The first one computes invariants for each component using a specific “analysis” method. The latter gathers all the invariants and uses them to prove properties by generating an SMT file containing the invariants’ conjunction and the safety properties’ negation as input to the Z3 SMT solver [69]. If the result is unsatisfied, the properties are invalid, and the Z3 solver returns a counter-example. Developers must refine the design and repeat the process until satisfactory results. The input of the iFinder tool consists of the following:

- \textit{BIP model and}
• safety properties that need to be proven\textsuperscript{4}. In particular, iFinder support only linear invariants.

4 NaturalBIP

The BIP framework supports exogenous coordination to build complex designs from smaller designs to enforce given properties. The challenge is applying BIP in designing cloud applications since cloud designers have no background knowledge of BIP. To develop an application in an exogenous approach, developers have to conduct a lot of manipulations that are difficult to learn (Fig. 1). This section introduces our language, NaturalBIP, for specifying behavioral constraints of the concurrent systems. Section 4.1 defines the ontology architecture to construct the NaturalBIP language. The architecture includes an ontology specifying system concepts and relationships between them, a component for describing conditions, and a domain-specific language for naturally writing functional requirements. The NaturalBIP Compiler translating the specification in NaturalBIP to JavaBIP artifacts and BIP connectors is presented in Section 4.2. To illustrate the feasibility of our proposed language, we use an example of Trackers-Peers communication, which is widely used in the existing work [57, 70].

Example 4.1 (Trackers-Peers communication). Consider the Trackers-Peers example inspired by a wireless audio protocol for peer-to-peer communication. There are two component types: Tracker and Peer. The protocol allows an arbitrary number of peers to communicate along a random number of wireless communication channels, and a unique tracker manages each channel. There is the list of requirements written in natural language as follows:

- Req\textsubscript{log}: "Every tracker shall log whenever a peer registering or unregistering to it."
- Req\textsubscript{action}: "Registered peers shall either speak to the channel or listen to other registered peers in the channel."
- Req\textsubscript{constraint}: "At most one registered peer shall speak at the moment."

4.1 NaturalBIP Language

We defined grammar to derive patterns that are pseudo-natural language templates with placeholders to capture requirements. Each derived pattern is associated with a formal representation in a logical language. By using these patterns, it addresses the ambiguity of natural language requirements. According to [? ], a boilerplate comprises attributes and fixed syntax elements. For example, in the boilerplate (3), “executes” is a fixed syntax element, while \langle instance \rangle and \langle action \rangle are placeholder attributes for user input.

\textsuperscript{4}In addition, one has to specify the analysis method used to infer invariants. We use atom-control and control-reachability (see [66] for details)
\langle\text{instance}\rangle \text{ executes } \langle\text{action}\rangle \quad (3)

We propose an ontology architecture for specifying the 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.

4.1.1 Ontology Architecture

Fig. 4 shows the architecture overview, which includes ontologies, components, and the relationships between them to preserve the semantics and resolve conflicts and ambiguities. The requirements follow the semantic definitions of NaturalBIP Requirement Ontology (NRO). The NRO imports the Domain Specific Ontology (DSO), transitively imports Behavioral Ontology (BO), and uses DSO instances.

Behavioral Ontology (BO) defines core concepts (e.g., \textbf{Subject}, \textbf{Action}, \textbf{State}, \textbf{Finite State Machine} (FSM)) that are used as requirements specification elements. It plays the role of top-level ontology, supporting broad semantic compatibility among domain-specific ontologies.

A Domain-Specific Ontology (DSO) contains the domain-specific classes of the system to be designed. DSO imports all classes in the top-level ontology (i.e., the BO) and further specializes in them. The DSO instances define corresponding instances or variables for the corresponding \textbf{Subject} in DSO.

The NaturalBIP Requirement Ontology (NRO) provides sentence structures using boilerplate forms with placeholders to construct a well-formed requirement, thus, decreasing specification errors.

Fig. 4: Ontology Architecture.
4.1.2 Behavioral Ontology (BO)

This ontology contains concepts specifying functional requirements and their relationships, such as components, actions, states, etc. These concepts are semantically interrelated, as shown in the conceptual overview in Fig. 5.

The BO concepts and their relationships are introduced and explained as follows:

- A **Subject** represents an interactive element in a concurrent system, and it has a finite state machine. Its status is determined by a State (via isIn property). A **Subject** can inherit from another **Subject** via the extends property and performs actions.
- Each finite state machine (FSM) contains a set of transitions (i.e., **TransitionSet**) and a set of states (i.e., **StateSet**).
- An **Action** denotes a behavior of a related **Subject**. One **Action** belongs to a **TransitionSet** in a FSM of the **Subject**. The execution of **Action** might change the **State** of the **Subject** (via sets property).
- Each **State** belongs to the **StateSet** of the **Subject**’s FSM and its value can be set by executing an **Action**.

4.1.3 Domain Specific Ontology (DSO)

DSO describes the vocabulary to express a system domain. The DSO concepts and interactions provide a semantic model of the system’s domain. Fig. 6 shows a DSO instance for the Trackers-Peers communication system conforms to Example 4.1.

There are two **Subjects** are **Tracker** and **Peer**. Each has a FSM, which contains a set of transitions and states. The **TransitionSet** of **Tracker** includes broadcast and log, while **Peer** consists of register, unregister, speak, and
State TrackerInit belongs to the StateSet of Tracker and states PeerInit, Registered belong to the StateSet of Peer.

4.1.4 NaturalBIP Requirement Ontology (NRO)

Fig. 7 shows the ontology for encoding elements of requirement specifications. The Requirement in our language is a Clause scoped by one or more Quantifiers (via hasClause and hasQuantifier properties). Each Clause always has a Main clause (hasMain property) and may have a Constraint (hasConstraint property). While the Main clause describes the occurrence of actions and may specify State Values scoped by corresponding Quantifiers, the Constraint clause specifies the observation of events and may describe State Values scoped by corresponding Quantifiers. Quantifier is a component determining the scope of application for specific classes in functional requirements. A Quantifier might contain Conditions. The Condition defines conditions or constraints that affect classes, instances, or the relations between them. Finally, Observing Event, Occurring Action, and State Value specify the observation of an event, the occurrence of an action, or the state’s value, respectively.

Quantifier, Condition, Occurring Action, State Value, and Observing Event are boilerplate syntax containing fixed elements and placeholders.
4.1.5 NaturalBIP Syntax and Semantics

The grammar rules of our language are presented in Table 1 with some notations defined in Table 2. The grammar rules are left-associative, and non-terminal symbols will be parsed until reaching terminal symbols. A Requirement is always scoped by at least one Quantifier and contains a Main clause. Constraint clause is optional in a Requirement, and if it exists, this clause will be separated from the Main clause by a comma symbol (i.e., “,”). As described in Table 1, Main is constructed by compound expressions including Occurring Actions, State Values, or Conditions with connective operators. If the Main clause and Constraint clause use a new variable, the new variable must be scoped by a Quantifier. Table 4 illustrates operators to associate elements in a compound expressions including: conjunction operator (i.e., “and”) and disjunction operator (i.e., “or”).

Table 3 presents a requirement’s derivable phrases if it contains constraints. The Constraint is defined after conjunction words (i.e., “if”, “while”, “after”, “before”), and the constraints expression consists of Observing Events, State Values, or Condition with connective operators. The Quantifier is optional in Constraint.

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.
Table 1: NaturalBIP grammar

(Requirement) ::= ((Quantifier))+ (Clause)

(Clause) ::= (Main) 
| (Constraint), (Main)

(Quantifier) ::= (quantification of amount) subject_instance, 
[(separator) (Condition),]

(Constraint) ::= (conjunction) (constraint expression)

(Main) ::= (compound expression)

(compound expression) ::= (compound expression) synchronized with (compound expression) 
| (compound expression) (connective) (State Value) 
| (compound expression) (connective) (Occurring Action) 
| (Quantifier) (State Value) 
| (Quantifier) (Occurring Action)

(constraint expression) ::= (constraint expression) (connective) (state value) 
| (constraint expression) (connective) (Observing Event) 
| (Quantifier) (State Value) 
| (Quantifier) (Observing Event)

(State Value) ::= instance is in state_value state

(Observing Event) ::= instance executes action 
| instance does not execute action

(Occurring Action) ::= instance shall action 
| instance shall not action 
| (instance shall either action_1 or action_2 or ... or action_n)

(Condition) ::= (Condition) (connective) (Condition) 
| instance is phenomenon [instances] 
| instance has phenomenon [instances] 
| instance can do phenomenon [instances]

(conjunction) ::= if | while | after | before

(connective) ::= and | or

(quantification of amount) ::= for all | for any | for every 
| there is a | there is one | there are some

(separator) ::= where | such that

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 (cf. Example 4.1).
Table 2: Notations used in our grammar

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨ . . . ⟩</td>
<td>Non-terminal symbol</td>
</tr>
<tr>
<td>⟨ . . . ⟩</td>
<td>Optional symbol</td>
</tr>
<tr>
<td>. . .</td>
<td>Disjunction logic operator</td>
</tr>
<tr>
<td>( . . . )+</td>
<td>The number of elements inside the parentheses is at least one</td>
</tr>
<tr>
<td><strong>bold string</strong></td>
<td>The placeholder with semantics defined or deduced from the DSO</td>
</tr>
<tr>
<td><em>emphasis string</em></td>
<td>Keyword</td>
</tr>
<tr>
<td>normal string</td>
<td>Normal meaning in the natural language</td>
</tr>
</tbody>
</table>

Table 3: The derivable Clauses contain Constraints

<table>
<thead>
<tr>
<th>Derivable Clause</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ⟨Constraint⟩, ⟨Main⟩</td>
<td>Globally, ⟨Main⟩ occurs next to the occurrence of ⟨Constraint⟩</td>
</tr>
<tr>
<td>While ⟨Constraint⟩, ⟨Main⟩</td>
<td>Globally, ⟨Main⟩ occurs during the observation of ⟨Constraint⟩</td>
</tr>
<tr>
<td>After ⟨Constraint⟩, ⟨Main⟩</td>
<td>⟨Main⟩ after ⟨Constraint⟩</td>
</tr>
<tr>
<td>After ⟨Constraint⟩, ⟨Main⟩</td>
<td>⟨Main⟩ after ⟨Constraint⟩</td>
</tr>
<tr>
<td>Before ⟨Constraint⟩, ⟨Main⟩ before ⟨Constraint⟩</td>
<td>⟨Main⟩ can be observed until the occurrence of ⟨Constraint⟩</td>
</tr>
</tbody>
</table>

Table 4: Grammar rules of Compound Expression

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>elem₁ and . . . and elemₙ</td>
<td>Globally, elements elem₁ to elemₙ occur at the same time.</td>
</tr>
<tr>
<td>elemₐ synchronized with elemₜ</td>
<td>Globally, the occurrence of elemₐ is synchronized with the occurrence of elemₜ.</td>
</tr>
<tr>
<td>elemₐ or elemₜ</td>
<td>At the current state, elemₐ or elemₜ or both can be observed.</td>
</tr>
</tbody>
</table>

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.

Quantifier. This element is mandatory to determine the application scope of the requirement. The quantifier is defined in the Quantifier rule in Table 1 and explained in Table 5. Looking at the requirement in Example 4.1, the quantifier for Peer can be presented as follows:

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Explanation</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨quantification of amount⟩</td>
<td>Mandatory. To specify the scope (universal or existential) for applying the requirement.</td>
<td>Universal quantifier: “all”, “any”, “every”</td>
</tr>
<tr>
<td>subject</td>
<td>Mandatory. Subjects affected by the quantifier.</td>
<td>Existential quantifier: “a”, “one”, “some”.</td>
</tr>
<tr>
<td>instance</td>
<td>Mandatory. The specific variable of the subject.</td>
<td>The value of this block is mapped from the corresponding Subject in the BO.</td>
</tr>
<tr>
<td>⟨separator⟩</td>
<td>Optional. Indicator to start describing conditions related to the quantifier’s instance.</td>
<td>“where”, “such that”.</td>
</tr>
<tr>
<td>⟨Condition⟩</td>
<td>Optional. It appears following separators to describe the condition(s) of variables.</td>
<td>This value can be inferred from the value of the Subject in BO and vocabulary in LO.</td>
</tr>
</tbody>
</table>

• “For all Peer p,” expresses that for all instance p of component Peer, or
• “There are some Peer p, such that p is registered to t,” expresses a condition applied for some instance p of component Peer, or
• “There is a Tracker t, where t is not null,” expresses another derivation of quantifiers with a condition but uses a different separator.

Conditions. This element describes the constraints or conditions to execute actions. The template of this block is defined in Formula (4):

\[
\text{instance is/has/can phenomenon [instances]} \tag{4}
\]

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 [71] and the
WordNet Lexical Database [72]. For examples, “p has no capacity”, “p is registered to t”.

Let us consider the requirements in Example 4.1, the functional requirements for Req_log can be written in our language as two following requirements:

- **TP_01_register**: “For any Peer p, there is a Tracker t, if p executes register, t shall log.”
- **TP_01_unregister**: “For any Peer p, there is a Tracker t, such that p is registered to t, if p executes unregister, t shall log.”

Fig. 8 illustrates the decomposition of the requirement TP_01_register. The requirement is decomposed following the defined grammar until the terminal symbol (i.e., Quantifier, Observing Event, Occurring Action, or Condition) is reached. The specific values (i.e., the bold strings in the grammar) are mapped or referred from the DSO and DSO instance. In particular, requirement TP_01_register has Main and Constraint clauses and scopes them by Quantifier-01 and Quantifier-02. The Constraint clause specifies the observation of event “p executes register”, and the Main clause specifies the occurrence of action “t shall log”. DSO instances contains the values deduced from the DSO’s concepts. Then, the placeholders use those values to complete the requirements.

![Diagram](image.png)

Fig. 8: The decomposition of requirement TP_01_register.

The original requirements Req_action and Req_constraints are related to the action speak of subject Peer and action broadcast of subject Tracker. Therefore, we write those requirements in NaturalBIP as follows:

- **TP_02_speak**: “For any Peer p, there is a Tracker t, such that p is registered to t, if t executes broadcast, p shall speak and for all Peer p1, where p1 is registered to t and p1 is different with p, p1 shall not speak.”
- **TP_02_listen**: “For any Peer p, there is a Tracker t where p is registered to t, if t executes broadcast, p shall listen or for all Peer p1, such that p1 is registered to t and p1 is different with p, p1 shall listen.”
In the case that there is more than one instance of Peer registered to the same Tracker’s instance, TP_02_speak states that at most one instance of Peer can speak, while TP_02_listern claims that more than one instance of Peer can listen in a specific moment. The three requirements have the same TP_02 part in their name because they associate each other following the description in Req_action and Req_constraint in Example 4.1.

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 the artifacts, including the JavaBIP macro, data transfers, and constraints from the functional requirements written in the NaturalBIP language. Fig. 9 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 get the information of Quantifier, Interaction, and Condition; and skolemizing the requirements (cf. Section 4.2.1).
- 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 [73], to generate CNF clauses (see Section 3.1).
- 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 [65].
- Step 5 BIP Generator: Generating artifacts, including JavaBIP macros, data transfers, conditions, and BIP connector in the form of the algebra of connectors (AC) [65].

Fig. 9: The process for the generation of JavaBIP artifacts from NaturalBIP requirements.
4.2.1 Pre-processing

Pre-processing converts the requirements written in NaturalBIP into input for PBL (i.e., Boolean formulas). This process is performed in the following steps:

1. **Pre-processing of NaturalBIP requirements**

   The output of the NaturalBIP compiler is the JavaBIP macro, which describes the interaction between system components through actions. Therefore, requirements written in the NaturalBIP contain descriptive information about some components’ states that must be transformed into the corresponding actions. Based on the FSM specification in the design, the transformation converts state description to the disjunction of actions leading to that state. For example,

   ![Finite state machines of Tracker and Peer](image)

   **Fig. 10**: The finite state machines of Tracker and Peer.

   To depict the use of some special expressions, consider a hypothetical requirement named **HTP** (i.e., hypothetical Trackers-Peers example) that specifies "For any Tracker $t$, while $t$ is in TrackerInit, there is a Peer $p$, where $p$ is registered to $t$, ($p$ shall either speak or listen) and $p$ shall not unregister.", where **Peer** and **Tracker** are components having the corresponding FSMs described in Fig. 10. State **TrackerInit** has two incoming transitions: **log** and **broadcast**. Thus, the state **TrackerInit** is replaced with two actions **log** and **broadcast** and "while" connective is replaced by "if" because "while" specifies the state values. The original requirement will be rewritten as "For any Tracker $t$, if ($t$ shall log or $t$ shall broadcast), there is a Peer $p$, where $p$ is registered to $t$, ($p$ shall either speak or listen) and $p$ shall not unregister." (cf. step 1 in Fig. 11).

   The expression "either ...or ..." in our language will be presented by "A XOR B" in Boolean formulas. The requirement **HTP** becomes "For any Tracker $t$, if ($t$ shall log or $t$ shall broadcast), there is a Peer $p$, where $p$ is registered to $t$, ($p$ shall speak XOR $p$ shall listen) and $p$ shall "unregister."" (cf. steps 2 and 3 in Fig. 11).

2. **Collecting components**

   After being pre-processed, the requirement is analyzed to extract quantifiers and actions following the corresponding grammar rules in Table 1 (cf. steps (1) and (2) in Fig. 12). To collect the conditions, first, remove the quantifiers and actions obtained in steps (1) and (2) from the request (cf. step (3) in Fig. 12). Then split the remaining string with the "and"/"or"
operators as keywords; remove the keywords (i.e., if, while) and parentheses to get the conditions (cf. step (4) in Fig. 12). Finally, actions and conditions are converted to variables for Boolean formulas and updated into the requirement (cf. step (5) in Fig. 12).

### 3. Skolemization

Skolemization is a process of removing quantifiers from a predicate logic formula. Assume that a requirement $REQ$ has an existential quantifier describing the instance $ei$ of a class $C$ and a universal quantifier describing instance $ui$ of class $C'$. If the existential quantifier is inside the universal quantifier, $ei$ will be replaced with a new value composed by concatenating the requirement name, instance name in the universal quantifier, and its
name (cf. equation(1.4)). In particular, $e_i$ will be replaced with $REQuiei$.

$$e_i = \begin{cases} REQuiei, & \text{if } e_i \text{ is bounded in } u_i. \\ e_i, & \text{otherwise.} \end{cases} \quad (5)$$

This way, the instance $p$ in the hypothetical example $HTP$ is replaced with $HTP_{tp}$. Thus, the $HTP$ becomes: “if (t\_log or t\_broadcast), $HTP_{tp\text{-registered-to}}t$ and ($HTP_{tp\text{-speak}}$ XOR $HTP_{tp\text{-listen}}$) and $HTP_{tp\text{-unregister}}$.” (cf. step (5) in Fig. 12).

Regarding the requirement $TP\_02$, which includes three related requirements $TP\_02\_mutex$, $TP\_02\_speak$ and $TP\_02\_listen$, its Boolean formulas after the pre-processing are shown in Listing 2:

Listing 2: $req\_file$ contains the Boolean formulas of requirement $TP\_02$

```
1 // File : TP\_02.txt
2 TP\_02\_speak = ((p\_speak =\> (p\_is\_registered\_to\_TP02\_speak\_ptt & p\_is\_different\_with\_p & ~p\_speak & p\_is\_registered\_to\_TP02\_speak\_ptt & TP02\_speak\_broadcast)))
3 TP\_02\_mutex = (p\_speak XOR p\_listen)
4 TP\_02\_listen = (( p\_listen =\> ( p\_is\_registered\_to\_TP02\_listen\_ptt & TP02\_listen\_ptt\_broadcast))) & ((p\_listen =\> ( p\_is\_registered\_to\_TP02\_listen\_ptt & p\_is\_different\_with\_p & p\_is\_registered\_to\_TP02\_listen\_ptt & TP02\_listen\_ptt\_broadcast)))
5 Main\_Exp: TP\_02\_speak & TP\_02\_mutex & TP\_02\_listen
```

4.2.2 Boolean Encoding

Boolean Encoding is the process of converting requirements containing Constraint clauses (i.e., “if”, “while”, “after”, and “before”) into Boolean formulas. Among grammar rules in Table 3, the pre-processing step converted “while” clause to “if” clause. Assume that the compound expression in Main contains a disjunction of $n$ sub-expression (i.e., $\bigvee_{i=1}^{n} expr_i$) and $expr_i$ are conjunctions of elements state value, observing event, occurring action, or Condition (as shown in Table 1). We denote the conjunctions as $\bigwedge_{j=1}^{\left|expr_i\right|} elm_{ij}$

We write $Elm$ to denote an element in the $Main$ clause. The grammar rules for “if”, “while”, and “after” are presented in formulas (6):

$$if/after\text{ Constraint, Main} \equiv if/after\text{ Constraint, } \bigvee_{i=1}^{n} \bigwedge_{j=1}^{\left|expr_i\right|} elm_{ij} \quad (6)$$

This pattern will be converted to Boolean formulas as equation (7):

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\left|expr_i\right|} (elm_{ij} \implies \text{Constraint} \land \bigwedge_{j' \neq j} elm_{ij'}) \quad (7)$$

As mentioned in Table 3, the grammar “Before $Constraint$, $Main$” states that “globally, $Constraint$ is absent in the occurrence of $Main$”, so the
NaturalBIP Language

The corresponding Boolean formula for this grammar is shown in formula (8):

\[
\bigwedge_{i=1}^{n} \bigwedge_{j \neq j'}^{|\text{expr}_i|} (\text{elm}_{ij}^i \implies \neg \text{Constraint} \land \bigwedge_{j' \neq j} \text{elm}_{ij'}^i) \tag{8}
\]

4.2.3 Dual-Horn clauses generation

Given a formula \( F \) in conjunctive normal form (CNF), where \( F = C_1 \land C_2 \land \cdots \land C_n \) and \( C_i = \neg x_1 \lor \cdots \lor \neg x_m \lor y_1 \lor \cdots \lor y_k \), where \( \neg x_j (j \in [1, m]) \) are negative literals and \( y_l (l \in [1, k]) \) are positive literals, the dual-Horn clauses are computed as follows:

1. For each clause \( C_i \) in \( F \):
   - If \( C_i \) contains at most one negative literal (i.e., \( m \leq 1 \)), keep \( C_i \)
   - Otherwise, split \( C_i \) into the disjunction of multiple clauses. \( C_i = C_i^1 \lor \cdots \lor C_i^m = (\neg x_1 \lor y_1 \cdots \lor y_k) \lor \cdots \lor (\neg x_m \lor y_1 \cdots \lor y_k) \)

2. Conjoin the resulting clauses with the logical “and” (i.e., \( \land \)) operator to obtain the final formula. Assume that clause \( C_1 \) contains more than one negative literal, the final formula \( F = (C_1^1 \land C_2 \land \cdots \land C_n) \lor \cdots \lor (C_1^m \land C_2 \land \cdots \land C_n) \) is the disjunction of dual-horn clauses.

After generating CNF clauses using the PBL library, the process to generate dual-Horn clauses [65] consists of the following steps:

1. **synthesize clauses having the same negative elements**

   Considering any two different clauses \( \text{cnf clauses}[i] \) and \( \text{cnf clauses}[j] \) in the given CNF formula, we denote by \( \text{cnf clauses}[i].\neg \) and \( \text{cnf clauses}[j].\neg \) the sets of negative variables and by \( \text{cnf clauses}[i].\pos \) and \( \text{cnf clauses}[j].\pos \) the sets of positive variables in \( \text{cnf clauses}[i] \) and \( \text{cnf clauses}[j] \), respectively. If two negative sets of \( \text{cnf clauses}[i] \) and \( \text{cnf clauses}[j] \) are equivalent (line 10 in Algorithm 1), assign \( \text{cnf clauses}[i].\neg \) to \( \text{temp clauses}.\neg \). The \( \text{temp clauses}.\pos \) is the Cartesian product of \( \text{cnf clauses}[i].\pos \) and \( \text{cnf clauses}[j].\pos \) (lines 12 and 13). If the \( \text{temp clauses}.\pos \) is not empty, add it into the \( \text{synthesized cnf clauses} \) (lines 17-19). For example, the two CNF clauses “(\( \neg x_1 \mid y_1 \mid y_2 \)) & (\( \neg x_1 \mid z_1 \mid z_2 \))” will be replaced by “(\( \neg x_1 \mid y_1 \& z_1 \mid y_1 \& z_2 \mid y_2 \& z_1 \mid y_2 \& z_2 \)).

2. **collect conditions to relevant negative actions**: We introduce a **causal rule** (Definition 4.1) that provides a mechanism for computing BIP connectors [65].

**Definition 4.1** (Causal rule [65]). A causal rule is a \( \mathbb{B}[P] \) formula \( E \implies C \), where \( E \) (the effect) is a constant true or an action variable \( p \in P \); and \( C \) (the cause) is a constant (true or false) or a disjunction of interactions, i.e., \( \bigvee_{i=1}^n a_i \) where, for \( i \in [1, n] \), \( a_i \) are conjunctions of action variables (i.e., \( a \in 2^P \)).
Algorithm 1 Combine clauses having the same negative variables

Require: cnf_clauses \(\triangleright\) given set of CNF clauses
Ensure: synthesized.cnf_clauses
1: synthesized.cnf_clauses := {}  
2: combined_clauses := {}  
3: i := 0 \(\triangleright\) counter variable for iteration
4: while i < cnf_clauses.size() do
5:     if cnf_clauses[i] ∈ combined_clauses then \(\triangleright\) ignore combined clauses
6:         continue
7:     end if
8:     initialize temp_clauses \(\triangleright\) create a temporary clause
9:     for j \(\in\) [i + 1, cnf_clauses.size()] do
10:        if cnf_clauses[i].neg = cnf_clauses[j].neg then
11:            temp_clause.neg := cnf_clauses[i].neg
12:            cnf_clauses[i].pos := cnf_clauses[i].pos × cnf_clauses[j].pos
13:            temp_clause.pos := cnf_clauses[i].pos
14:            combined_clauses := combined_clauses ∪ {cnf_clauses[j]}
15:        end if
16:     end for
17:     if temp_clause.pos.size() != 0 then
18:         synthesized.cnf_clauses := synthesized.cnf_clauses ∪ temp_clause
19:     end if
20:     i := i + 1
21: end while

There are three types of causal rules with constants:
(a) true ↦ a expresses that the considered interaction contains a,
(b) The form p ↦ true are satisfied by all interactions, and
(c) p ↦ false expresses that the considered interaction does not contain p.

Definition 4.2 (Interaction). An interaction \(a \in 2^P\) satisfies a formula \(R \in \mathbb{B}[P]\) (denoted \(a \models R\)) iff the corresponding boolean valuation satisfies \(R\). A term \(x \in AI(P)\) (i.e., algebra of interactions \([64]\)) satisfies \(R\) (denoted \(x \models R\)) iff all interactions belonging to \(x\) satisfy \(R\), that is
\[
x \models R \overset{\text{def}}{\iff} \forall a \in x, a \models R
\]

A non-constant causal rule can be represented in the form \(\overline{p} \lor \bigvee_{i=1}^n a_i\) where, p is a variable belonging to the action set \(P\) of the whole system (i.e., \(p \in P\)), and \(a_i\) with \(i \in [1, n]\) are conjunctions of action variables (i.e., \(a \in 2^P\)). Therefore, if one of the negative variables in a clause is a condition, it must be combined with the related negative action by De Morgan’s laws \([74]\).

For example, consider the clause “\(\sim p\text{\_speak} \lor \sim p\text{\_listen} \lor \sim p\text{\_is\_registered\_to\_t}\)”, where “p\_is\_registered\_to\_t” is a condition and the action “p\_speak” is related to the condition “p\_is\_registered\_to\_t” because they have the same instance value “p”. Thus, they must be combined to have a new clause “\(\sim (p\text{\_speak} \& \ p\text{\_is\_registered\_to\_t}) \lor \sim p\text{\_listen}\)” which presents two actions in the corresponding causal rule.
3. generate the sets of dual-Horn clauses

Dual-Horn clauses are the disjunctions of variables where at most one is negative and, consequently, consists of non-constant causal rules. Since each dual-Horn clause represents a causal rule, a dual-Horn set is a system of causal rules defined in Definition 4.3 [65].

**Definition 4.3** (System of Causal rules). A system of causal rules is a set \( R = \{ p \Rightarrow x_p \}_{p \in P^t} \), where \( P^t \overset{def}{=} P \cup \{ \text{true} \} \). An interaction \( a \in 2^P \) satisfies the system \( R \) (denoted \( a \models R \)), iff \( a \models \bigwedge_{p \in P^t}(p \Rightarrow x_p) \). We denote by \( R \) the union of interactions satisfying \( R \):

\[
R \overset{def}{=} \sum_{a \models R} a
\]

The causal rules can be simplified as follows:

\[
\{ p \Rightarrow a_1, p \Rightarrow a_2 \} \leadsto \{ p \Rightarrow a_1 \land a_2 \}
\]

Furthermore, notice that \( a_1 \lor a_1a_2 = a_1 \), thus causal rules can be also simplified accordingly:

\[
(p \Rightarrow (a_1 \lor a_1a_2)) \leadsto (p \Rightarrow a_1)
\]

All the causal rules are simplified and absorbed using (9) and (10). Each system of causal rules contains constant (cf. Definition 4.1), including \( \text{true} \Rightarrow \bigvee_i p_i \) and \( p_i \Rightarrow \text{true} \), where \( p_i \) are actions in the considered system of causal rules. Listing 3 presents a system of causal rules after adding the causal rules with constant.

**Listing 3:** The system of causal rules after adding causal rules with constant

```plaintext
1 { p\_listen => TP02\_listen\_ptt
2   broadcast,
3   p\_1\_listen => TP02\_listen\_ptt\_broadcast,
4   p\_listen => true, p\_1\_listen => true,
5   TP02\_listen\_ptt\_broadcast => true,
6     true => p\_listen \_ p\_1\_listen \_ TP02\_listen\_ptt\_broadcast
7 }
```

4. saturate the systems of causal rules: The systems of causal rules need to be saturated to construct BIP connectors.

**Definition 4.4** (Saturated System of Causal Rules [65]). A system of causal rules \( \{ p_i \Rightarrow x_i \}_{i=1}^n \) is saturated iff, for all \( i \in [1,n] \), \( x_i = x_i[(x_jp_j)/p_j] \), where \( x_i[(x_jp_j)/p_j] \) is obtained by substituting \( (x_jp_j) \) for \( p_j \) in \( x_i \), for all \( j \neq i \).

For example, consider a system of causal rules \( \mathcal{CR}(P) = \{ p \Rightarrow a_p, q \Rightarrow qa_q \} \), where \( p,q \in P \) are ports, and \( a_p,a_q \in 2^P \) are interactions.
We obtain the saturated system of causal semantic $\mathcal{CR}_{sat}(P) = \{ p \Rightarrow a_p, q \Rightarrow p_a p a_q \}$ by substituting $p a_p$ for $p$ in the second rule. Obviously, $\mathcal{CR}(P)$ and $\mathcal{CR}_{sat}(P)$ are equivalent:

$$
(p \Rightarrow a_p) \land (q \Rightarrow p a_q) = (\overline{p} \lor a_p) \land (\overline{q} \lor p a_q)
$$

$$
= \overline{p} \overline{q} \lor a_p \overline{q} \lor p a_p a_q = \overline{p} \overline{q} \lor a_p \overline{q} \lor p a_p a_q a_p
$$

$$
= (\overline{p} \lor a_p) \land (\overline{q} \lor p a_p a_q) = (p \Rightarrow a_p) \land (q \Rightarrow p a_p a_q)
$$

Listing 4 shows the saturated system of causal rules for the corresponding one in Listing 3.

Listing 4: Saturated system of causal rules

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\ p_listen \ =\ =&gt; p_listen \ &amp; \ TP02listenptt_broadcast,</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\ p1_listen \ =\ =&gt; p1_listen \ &amp; \ TP02listenptt_broadcast,</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>\ TP02listenptt_broadcast \ =\ =&gt; true,</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>\ true \ =\ =&gt; TP02listenptt_broadcast</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.4 JavaBIP artifacts generation

JavaBIP macros represent BIP connectors in Java implementation to coordinate the components’ activities at runtime. Algorithm 2 presents how to generate the macro codes from the dual-Horn clauses representing the systems of causal rules in Listing 3.

For each clause in the considered dual-Horn set (lines 5 and 6 in Algorithm 2), 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 “$true \equiv \lor_{i=1}^{n} p_i$” (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).

Listing 5 explains the process of generating “accepts_macro”. From the set of actions (line 2), we replace instances names with classes name (line 5) and then generate the “accepts” part for the JavaBIP macro (lines 8-10).

Listing 5: Accept parts generated from the set of actions

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\ //accepts --- set of actions</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\ accepts = { ‘TP02listenptt_broadcast’, ‘p_speak’, ‘TP02speakptt_broadcast’, ‘ p_listen ’, ‘ p1_listen ’ }</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>\ // replace instances names with classes name</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm 2 JavaBIP macro generation

1: Input: dualHorn_clauses ▷ sets of dual-Horn clauses
2: Output: javaBIP_GlueBuilder ▷ JavaBIP macros
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 ▷ This is a non-constant clause
9:         requires := requires ∪ {clause}
10:     else ▷ This clause contains a set of interactions
11:        accepts := accepts ∪ clause.pos
12:     end if
13:   end if
14: end for
15: //replace instance names with corresponding class names in requires and accepts
16: for clause ∈ requires do
17:   require_macro += “port(“+clause.neg+“).requires(“+“,”.join(clause.pos)+“);”
18: end for
19: for elm_i ∈ accepts do
20:   accepts_macro += “port(“ + elm_i + “).accepts(“ + “,”.join (accepts \ {elm_j}) + “);”
21: end for
22: return require_macro + accept_macro

7 // generate corresponding accepts_macro
8 port(Peer.class,"listen").accepts(Peer.class,"speak",Peer.class,"listen",Tracker.class,"broadcast");
9 port(Peer.class,"speak").accepts(Peer.class,"listen",Tracker.class,"broadcast");
10 port(Tracker.class,"broadcast").accepts(Tracker.class,"broadcast",Peer.class,"speak",Peer.class,"listen");

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 “⟨instance1⟩ is/has/can something ⟨instance_i⟩” (where i ≥ 0) and the subject of ⟨instance1⟩ is different from the subject of ⟨instance_i⟩, there is a data transfer from ⟨subject_i⟩ to ⟨subject1⟩. 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” which is for “ TP02speaktp.is_registered_to_t , TP02listentp.is_registered_to_t , p1_is_registered_to_t ”.

As mentioned in Section 3.2.1, JavaBIP classes contain @Guard functions specifying the condition to fire FSM’s transitions. Each condition is considered a @Guard (i.e., a boolean function). The boolean functions are generated as a template with the default returning value set as “True”. Depending on the context of the requirements, developers write the actual content for those functions. The parameters of those functions are generated depending on the kind of ⟨subject_i⟩:
• If ⟨subject_i⟩ is empty, there is no parameter,
• If ⟨subject_i⟩ is the same as ⟨subject1⟩, generate a normal parameter,
In `Peer.java`:

```java
@Guard(name="is different from")
public boolean is different from (Peer Peer_ins) {
    return true;
}
```

- If `<subject_i>` is different from `<subject_1>`, the generated parameter is a `@Data` received through the specified data transfers.

In `Peer.java`:

```java
@Guard(name="is registered to")
public boolean is registered to (@Data(name="Tracker2Peer.data") Tracker Tracker_ins) {
    return true;
}
```

### 4.2.5 BIP Connectors generation

Algorithm 3 describes the BIP connectors generation. This algorithm is a variation of the algorithm constructing a causal interaction tree from a saturated system of causal rules in [65]. Thus, the correctness argument for our algorithm is similar. The input of Algorithm 3 is a set of atomic interactions computed from the system of causal rules. Let \( X = \{ p \mapsto x_p \}_{p \in P} \) be a saturated system of causal rules, with \( x_p = \bigvee_{i=1}^{n_p} a_i^p \), atomic_interaction_set = \( \{ pa_p^i | p \in P \cup \{ \text{true} \}, i \in [1, n_p] \} \). The output is BIP connectors, which are the textual representation of \( \mathcal{AC}(P) \).

#### Algorithm 3 BIP connectors generation

<table>
<thead>
<tr>
<th>Input: <code>ais</code></th>
<th>set of atomic interactions computed from dual-Horn clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: <code>bip_connector</code></td>
<td>BIP connectors formalized in ( \mathcal{AC}(P) )</td>
</tr>
</tbody>
</table>

1. procedure `genConnector(ais)`
2. initialize `triggers`, `remains`, `result_str`
3. triggers = \{ `ais_i` | card(`ais_i`) = 1 \}
4. remains = `ais` \ `triggers`
5. triggers = `triggers` \& `get_intersection(remains)`
6. if `triggers` = \( \emptyset \) then
7.     `result_str` += `gen_rendezvous_str(remains)`
8. else
9.     `result_str` += `gen_triggers_str(triggers)`
10. end if
11. for each `trigger` in `triggers` do
12.     `dependent_elements` = `get_dependent_elements(remains, trigger)`
13.     if `is_connector(dependent_elements)` then
14.         `result_str` += `genConnector(dependent_elements)`
15.     else
16.         `result_str` += `gen_rendezvous_str(dependent_elements)`
17.     end if
18. end for
19. `return result_str`
20. end procedure

Consider a set of interactions \( \{p, pq, pqrt\} \), and each element is a set of ports. Thus, the input for the algorithm is "`ais = \{\{p\}, \{p, q\}, \{p, q, r, t\}\}". In the first steps, the algorithm detects and separates the triggers
(triggers) (i.e., “\{p\}”) and the remaining elements (remains) (i.e., “\{p,q\}, \{p,q,r,t\}\}”), where the triggers are sets with a single element (lines 3 and 4).

The function \texttt{get\_intersection(remains)} takes input as the remains to compute an interaction that is a sub-interaction of all remains’ elements. If the computed one is not empty and does not contain elements in triggers, it will be added into triggers (line 5). In this example, although the intersection of “\{p,q\}” and “\{p,q,r,t\}\}” is “\{p,q\}\}”, triggers contains “\{p\}\}”. Thus, triggers does not add “\{p,q\}\}” into it.

Considering triggers, if triggers is empty, the input presents a Rendezvous connector (lines 6 and 7). The function \texttt{gen\_rendezvous\_str(remains)} generates the textual representation of the Rendezvous connector. For instance, if \texttt{triggers = \{\}} and \texttt{remains = \{\{p,q\}\}}, \texttt{result\_str} is “\texttt{(p)-(q)}” (i.e., \texttt{pq} in \texttt{AC(P)}). Otherwise, generating the textual representation of triggers (line 9). At this step, because \texttt{triggers = \{\{p\}\}} the \texttt{result\_str} appends “\texttt{(p)}” using function \texttt{gen\_triggers\_str(triggers)}.

In the next step, the function \texttt{get\_dependent\_elements(remains, trigger)} calculates elements in remains which are dependent on each element in triggers (line 12). For example, for \texttt{triggers = \{\{p\}\}} and \texttt{remains = \{\{p,q\}, \{p,q,r,t\}\}}, the function returns \{\{q\}, \{q,r,t\}\}. If \texttt{dependent\_elements} is a connector (line 13), invoke the algorithm recursively with the input is \texttt{dependent\_elements} (line 14). In particular, the function \texttt{is\_connector(dependent\_elements)} checks whether the intersection of any two elements in \texttt{dependent\_elements} is not empty. If it is not empty, \texttt{dependent\_elements} is considered a set that constructs a connector. In this example, \texttt{dependent\_elements} is a connector because the intersection of \{q\} and \{q,r,t\} is \{q\} (i.e., not empty). Thus, we run the algorithm recursively on the set \{\{q\}, \{q,r,t\}\}. Following the above steps, there is another trigger (i.e., “\texttt{(q)}”) and the \texttt{dependent\_elements} = \{\texttt{′r′}, \texttt{′t′}\}. If \texttt{dependent\_elements} is not a connector (line 15), generate a textual representation for \texttt{synch\_on\_elements} as a Rendezvous connector (line 16). As a result, the BIP connector of \{\{q\}, \{q,r,t\}\} is \texttt{q}'[rt] (the corresponding textual representation is “\texttt{(q)}'[-[\texttt{(r)}-[\texttt{(t)}]]}”) and the final BIP connector is \texttt{p'[q'[rt]]} (the corresponding textual representation is “\texttt{(p)}'[-[\texttt{(q)}'-[\texttt{(r)}]-[\texttt{(t)}]]}”).

Listing 6 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 in Listing 4.

Listing 6: Generated BIP connectors

```
1     // sets of atomic interactions
2     {{\"TP02\listen\_ptt\.broadcast\"}, {{\p\_listen \', \TP02\listen\_ptt\_broadcast \'}}, {{\p\_\listen \', \TP02\listen\_ptt\_broadcast \'}}}.
3     {{\p\\_\listen \', \TP02\listen\_ptt\_broadcast \'}, {\p\_\speak \', \TP02\speak\_ptt\_broadcast \'}}, {\p\_\listen \', \TP02\listen\_ptt\_broadcast \'}, {\TP02\speak\_ptt\_broadcast \'}}},
4     {{\p\_\listen \', \TP02\listen\_ptt\_broadcast \'}, {\TP02\listen\_ptt\_broadcast \'}}}
5
6     // corresponding generated BIP connectors
7     (\Tracker\_.\broadcast\_)−(\Peer\_.\listen\_)−(\Peer\_.\listen\_)
8     (\Tracker\_.\broadcast\_)−(\Tracker\_.\broadcast\_)−(\Peer\_.\speak\_)−(\Peer\_.\listen\_)
```
5 OCCIwareBIP framework

As modern software systems, cloud applications are inherently concurrent since these applications’ components usually run simultaneously and share access to the resources. OCCIware is a model-driven environment to model types of resources and assembly of resources’ instances. After modeling an application using OCCIware design, as shown in Fig. 13, the developer can create a configuration to specify and monitor instances. The OCCIware provides a means for specifying the behavior associated with OCCI entities as Finite State Machines (FSM). Although such FSMs can be used to monitor and coordinate the behavior of the corresponding entities, no such mechanisms are currently available. Furthermore, the OCCIware design is unable to describe the behavioral constraints, which specify how the software should behave or operate in certain situations.

We proposed a new framework named OCCIwareBIP, which integrates JavaBIP—a framework for the exogenous coordination of concurrent Java components into OCCIware—a framework for designing cloud applications. An executable model is generated to verify whether the design is deadlock-free. This section describes the workflow of the OCCIwareBIP toolchain, which extends the OCCIware framework to allow the safe management of resources at run time, as shown in Fig. 14. The light blue boxes show pre-existing artifacts from the OCCIware framework we extended; the dark blue ones are developed in this work.

To assist OCCIware architects in designing cloud applications following the exogenous approach, we extend the OCCIware Metamodel described in [75] by
naturalBIP Language

Fig. 14: Model-Driven Managing Everything as a Service with OCCIwareBIP.

defining different concepts required to model components and the coordination between them (Section 5.1). Section 5.2 describes algorithms to prepare the BIP model and the “.inv” file (i.e., instructions to compute the system’s invariants) for the verification using the iFinder model checker.

5.1 Concepts for extending coordination capability in the OCCIware design

The OCCIware design supports developers in specifying cloud components and their structural constraints. In contrast, behavior constraints are implemented by developers in the implementation phase. With this approach, the developers must track and analyze impacted codes whenever behavior constraints change, and this process is time-consuming. To quickly adapt to those changes, we defined the concept “Specification” specifying those constraints in design.

Concepts for specifying behavioral constraints.

Fig. 13 illustrates the OCCIwareBIP design of a Web application named Trackers-Peers, where Tracker and Peer are components, and Specification is the new concept specifying the coordination between those components. Components’ behaviors can be represented by an FSM, consisting of a finite
set of states and a finite set of transitions between these states. The transitions are associated with the corresponding actions of each OCCIware component.

Listing 7: Requirements are specified using “Specification”

```
<annotations name="Specification">
  <annotation id="TP_01_1">For any Peer p, there is a Tracker t, if p executes register, t shall log.</annotation>
  <annotation id="TP_01_2">For any Peer p, there is a Tracker t, if p is registered to t and p executes unregister, t shall log.</annotation>
  <annotation id="TP_01_3">For any Peer p, there is a Tracker t, if t executes log, p shall register or p shall unregister.</annotation>
  <annotation id="TP_02_speak">For any Peer p, there is a Tracker t such that p is registered to t, if t executes broadcast and p executes speak, for all Peer p1 such that p1 is registered to t and p1 is different with p, p1 shall not speak.</annotation>
  <annotation id="TP_02_listen">For any Peer p, there is a Tracker t where p is registered to t, if t executes broadcast, p shall listen or for all Peer p1, such that p1 is registered to t and p1 is different with p, p1 shall listen.</annotation>
  <annotation id="MAIN">TP_01, TP_02</annotation>
</annotations>
```

Listing 7 details the Specification. Each annotation specifies:

- A functional requirements written in our NaturalBIP language (lines 2-6);
- Annotation “MAIN” indicates the selected requirements from the list of requirements (line 7). To select all the requirements, the value of “MAIN” can be set as “all”. To select all except a small set of requirements, “MAIN” is written in the template “except requirement_i”, where requirement_i are deselected requirements.

**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 (see Section 4.1.2) as shown in Table 6.

**Table 6**: Mapping between the Behavioral Ontology classes and the OCCIware metamodel concepts

<table>
<thead>
<tr>
<th>OCCIware metamodel concepts</th>
<th>Behavioral Ontology classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>Subject</td>
</tr>
<tr>
<td>Mixin</td>
<td>Subject</td>
</tr>
<tr>
<td>FSM</td>
<td>FSM</td>
</tr>
<tr>
<td>Transition</td>
<td>Transition</td>
</tr>
<tr>
<td>State</td>
<td>State</td>
</tr>
<tr>
<td>Action</td>
<td>Action</td>
</tr>
</tbody>
</table>

Fig. 15 illustrates our DSO for the cloud domain, which is the semantic model for writing specifications in the NaturalBIP language, following the mapping in Table 6.
Concepts for specifying action types.

When extending the coordination capability of the OCCIware design, actions in the OCCIware design should be able to describe observed events (i.e., spontaneous), reactions when observing the event (i.e., enforceable), or internally updates the component’s states (i.e., internal). Therefore, we update the OCCI Metamodel by defining a new property named “actiontype” to specify each action type (Fig. 16).

When declaring actions in the OCCIware design, if the value of “actiontype” is spontaneous or internal, the transition is spontaneous or internal,
respectively. The transition is enforceable if the value of "actiontype" is enforceable or empty (by default).

5.2 Generate artifacts for verification

The design can be verified from the specified behavioral constraints to check whether it satisfies some safety properties 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 computing 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”

Options -at and -ct represent atom and compound types, respectively. Option -a specifies the applied analysis method (see Section 3.2.2).

The BIP model is generated from the OCCIwareBIP design and the OCCI configuration model. Consider connector $c = (s.\text{act})' -[(t.\text{act})'-(v.\text{act})]$ in Fig. 17, let the configuration model $\text{config\_model} = \{s:\{s_0, s_1\}, t:\{t_0, t_1\}, v:\{v_0\}\}$ specify instances of each class $s$, $t$, $v$. Finally, the concrete connectors will be generated as a list of all possible combinations of component instances in the configuration model as shown in Listing 8.

![Fig. 17: Connector c in the tree structure.](image)

Listing 8: generated BIP connectors in the BIP language for connector c

```plaintext
1  connector type connector_c1_define (Port p1, Port p2)
2      export port Port ep() 
3      define p1' p2
4  end
5
6  connector type connector_root_define (Port p1, Port p2)
7      define p1' p2
8  end
9
10  compound type connectorCompound()
11      component t t0, t1
12      component s s0, s1
13      component v v0
14
15  connector connector_c1_define connector_c1_0(t0.\text{act}, v0.\text{act})
16  connector connector_c1_define connector_c1_1(t1.\text{act}, v0.\text{act})
17  connector connector_root_define connector_root_0(s0.\text{act}, connector_c1_0.ep)
18  connector connector_root_define connector_root_1(s0.\text{act}, connector_c1_0.ep)
```
5.3 Integration of JavaBIP into OCCIware implementation

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

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

The coordination code that describes the interaction between the components and the data exchange is the \texttt{GlueBuilder} class, which is generated following Section 4.2.4.

In [65], 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. They are, therefore, also equivalent among themselves. Thus, verification results obtained on the BIP model remain valid for the JavaBIP one.

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. Instead of learning to write BIP connectors (e.g., \texttt{connector c = (s.act)’ – (t.act)’ – (v.act)} in Fig. 17), BIP models\(^5\) for the verification and JavaBIP macros\(^6\) for the implementation manually, our toolchain supports generating those artifacts automatically from the specifications written in NaturalBIP language (Listing 7).

Table 7 compares the number of lines needs to specify the functionality, which is choosing the third policy. Instead of writing 4127 lines for specifying the requirement using BIP connectors, writing verification code in the BIP model, and JavaBIP macros in the implementation, developers need only 5 lines to specify the requirement in our NaturalBIP language. In particular, the number of possible connectors in the BIP model increases exponentially with the number of components. In the case of 7 instances, including 3 Trackers and 4 Peers, there are 396 connectors (each connector is written in one line). When

\(^5\)https://github.com/TrinhLK/NaturalBIP-Compiler/blob/master/test/gen-data/trackerpeer.bip
\(^6\)https://github.com/TrinhLK/NaturalBIP-Compiler/blob/master/test/gen-data/JavaBIP_GlueBuilder.txt
the number of components increases to 75, there are 1,17·10^{24} connectors [57]. In another case, the Trackers-Peers example has 1,17·10^{24} connectors for 75 components, as reported in [57]. Meanwhile, the number of NaturalBIP specifications is kept the same.

### 6 Conclusion

We have provided a proof-of-concept implementation of our approach focusing on wide-sparing the application of the BIP framework in software development. Based on the ontology-driven requirement engineering and semantic analysis approach, we proposed NaturalBIP—a domain-specific language to specify the functional requirements for developing exogenously concurrent software. In this language, we defined grammar rules and templates with ontology-based semantics (Section 4.1.5) to tackle the ambiguity of the natural language syntax and express the semantic relationships and implicit assumptions through the information in the NRO (Section 4.1.4). Furthermore, we have shown how to develop and integrate domain-specific ontologies (Section 5.1).

We also proposed the NaturalBIP compiler for interpreting the specifications written in our NaturalBIP language to generate: (1) the BIP model—a formal model for verifying whether the design system satisfies safety properties using iFinder (Section 4.2.5); (2) Java classes with JavaBIP annotation: Since each Java class represents a component, it uses JavaBIP annotations to describe the component’s behaviors through its FSM; and (3) JavaBIP GlueBuilder, which implements the synchronization as described in the requirements written in JavaBIP (Section 4.2.4). This compiler also generates conditions (transition guards) as Boolean functions and handles some basic errors, such as undefined actions or states of classes from the requirement written in the NaturalBIP language.

The effectiveness of the overall approach was evaluated by introducing the OCCIwareBIP framework and developing the Trackers-Peers example. Using the NaturalBIP language saves time and effort verifying and implementing the target systems following the exogenous approach. The advantage of the exogenous approach is applied by selecting or modifying the specification list (Section 5.1).
In prospects, our interest lies in validating our ontology for the overall coverage of the cloud domain. Some analysis tools allow evaluating an ontology in both qualitative and quantitative aspects [76–81], such as the depth of inheritance tree for classes or their ancestor, etc., or identifying potential problems of an ontology. Furthermore, our methodology is not limited to OCCLware Studio for developing cloud applications, and it can be applied to any design framework supporting the FSM specification. Using our method on other design platforms in different contexts is also a potential direction.

Declarations

**Ethical Approval**
Not applicable.

**Competing Interests**
I declare that the authors have no competing interests as defined by Springer or other interests that might be perceived to influence the results and/or discussion reported in this paper.

**Author Contributions**
Writing - original draft preparation: Trinh, Le-Khanh; Writing - review and editing: all authors.

**Funding**
The author(s) received no financial support for this article’s research, authorship, and/or publication.

**Availability of data and materials**
All data and source codes are available.

**References**


