

CHAFIKA DJAOUI  
ALLAOUA CHAOUI

## FORMALIZATION AND ANALYSIS OF UML 2.0 INTERACTION OVERVIEW DIAGRAM USING MAUDE REWRITING LOGIC LANGUAGE

**Abstract** *The visual modeling language UML embodies object-oriented design principles. It provides a standard way to visualize the design of a system. It exploits a rich set of well-defined graphical notations for creating abstract models. However, the power of UML is lessened through partially specified formal semantics. Indeed, UML notations are semi-formal and do not lead to fully formalized and executable semantics. Fortunately, UML diagrams are prone to early formalization. Formal methods are a valuable tool that can help overcome the UML constructs' shortage of firm semantics. It is a powerful way to ascribe precise semantics to the graphical notations used in UML diagrams and models. Our work aims to support the semantics of the UML Interaction Overview Diagram. It introduces an approach to leveraging the strengths of the Maude Rewriting Logic language as a formal specification language. The proposal relies on a model-driven engineering approach. It aims to automate the UML Interaction Overview Diagram's mapping to a Maude language specification. The Maude language and its linked tools, including the Maude Model Checker, are used to analyze and verify the resulting Maude specification. Finally, an application example shows the feasibility and benefits of the proposed approach.*

**Keywords** UML Interaction Overview Diagrams, formal methods, rewriting logic, Maude language, Model-Driven Engineering (MDE), EMF, Sirius, Acceleo

**Citation** Computer Science 25(3) 2024: 397–419

**Copyright** © 2024 Author(s). This is an open access publication, which can be used, distributed and reproduced in any medium according to the Creative Commons CC-BY 4.0 License.

## 1. Introduction

Since its outset, the Object Management Group UML standard has become a general-purpose modeling language and a vital component of the Object-Oriented systems industry [31,32]. It has considerably influenced the specification and development of object-oriented software systems. UML offers a standardized and visual approach to portraying the design and architecture of software systems. It encompasses a variety of simple graphical and textual representation techniques in different phases of the system development cycle. These representation techniques allow the construction of abstract representations (called models), which facilitate the development process and reduce the complexity of the produced system [25].

UML is a semi-formal language rich in syntax and imprecise in semantics. The software developers cannot deny the significant gap between syntax and semantics in the UML-built models. Although the semi-formal nature of UML is an upbeat factor for its convenience and practicality [6], the correctness checking of models requires a firm semantics foundation. The solid semantics foundation caters to a rigorous study of UML diagrams and provides improved accuracy in reasoning about their properties. Nowadays, systems are getting more complicated and need ways to predict problems early in the development process. Anticipating fault detection leads to a successful system specification with low cost and high quality. Hence, it is advisable to map the semi-formal UML notations to concise notations to inspect efficiently whether a model meets the designer's intentions [7]. Formal methods and mathematical techniques are the more powerful approaches that lead to efficient ways to enhance the analytical capabilities of the UML notations.

The OMG has supervised a massive revision of UML 1.X [12], to enlarge the language expressiveness and relevance [34]. Indeed, UML has been revised noticeably through its successive versions (it has now reached version 2.5). UML Version 2.0 has improved the control flow views [18] through a new variant of interaction diagrams called Interaction Overview Diagram (IOD).

An IOD is a two-level behavioral diagram that covers the overall concepts and notations of the Activity Diagram (AD). It highlights interactions within the system and illustrates the control flow at a high level of abstraction. It is a variant of the AD that entails various components, such as interactions, control nodes, and decision points at the top level, also known as the overview level. However, the interaction level (lower level) incorporates detailed interaction diagrams that enable more explicit venturing of specific interactions when needed. An IOD serves as a workflow, business process viewer, or use case that requires more than one interaction diagram to represent multiple flows within a system. Therefore, an IOD is useful for decomposing and modeling complex scenarios that entail the representation of many alternatives in a single diagram. At variance to AD, IOD makes explicit which objects or actors perform activities and how they exchange messages with each other. Being a semi-formal language, IOD comes with imprecise semantics that can lead to misinterpretation,

faulty comprehension, and errors. Hence, it is necessary to translate the IOD into a precise specification that can be verified and analyzed through execution.

This work presents an approach to assigning formal semantics to the IOD. It introduces a general semantic framework for formalizing the IOD in a rewriting logic-based formalism. Accordingly, the IOD core constructs are staffed with precise and formal definitions. We exploit Rewriting Logic (RL) and the Maude language as a firm notation that supports the formal specification of a range of modeling languages [26]. Maude and its associated mathematical tools emerge as a three-part benchmark: a declarative programming language, an executable specification language, and a formal verification framework [10]. Indeed, the Maude tool is suitable for formal specification and early model correctness analysis [9]. In this paper, we propose a mapping from the semi-formal IODs notations into a mathematically equivalent specification in the Maude language. The Maude specifications allow for either execution through simulation or verification using the underlying Maude tools. To reach our goal, we propose to use the model-driven engineering (MDE) approach that relies on meta-modeling and model transformations. We employ well-known standards and tools on the Eclipse platform to automate our approach. Our automatic transformation-based approach has the advantage of reducing design errors at the early stage of software development with low cost and time efficiency. Concretely, our work includes the following contributions:

- We propose a simplified EMF meta-model for IODs. Then, we generate a visual modeling environment to edit and manipulate instances of the IOD's meta-model.
- We define Aceleo templates for mapping IOD notations to a corresponding Maude specification, ensuring automatic Maude code generation.
- We implement a Maude-language executable semantic framework. Hence, we validate the correctness of the semantics of the specified IODs by simulation or verification using the Maude model checker

The rest of the paper follows the upcoming sections: Section 2 summarizes an overview of related works. Section 3 addresses the IOD's key concepts. We recall RL and the Maude language in Section 4. The ensuing section outlines the IOD formalization using Maude language as a mathematical framework. In Section 6, we describe the MDE-based approach. Next, we delineate the implementation of the Case study. The last section concludes the paper and gives some perspectives on this work.

## 2. Related works

UML is widely used as a versatile modeling language for system specification, with a broad scope to represent different domains. However, the UML semantics are informal and unclear, leading to varied interpretations. Strict verification and tool support are only limited to syntactic issues [19]. The previous concerns led the UML formalization research stream to define the hidden semantics under UML diagrams using formal methods that bridge the gap between UML syntax and semantics.

Due to this, many works have attempted to combine the strengths of formal methods and UML flexibility.

In the literature, most of the approaches have shared the idea of transforming UML models into semantic domains that own verification tools such as Petri Nets [4,5]. SPIN and its underlying verification tool PROMELA, are worth citing [23]. Unlike other UML diagrams, the IOD has not been extensively covered. A few research studies dealt with its formalization. To address the semantic deficiency of the IOD authors have exploited the stochastic process algebra PEPA nets in [21]. Whittle [36] has developed a simulation environment for building and evaluating the semantics of IOD structures using hierarchical finite state machines. The research work in [1] has described a method that transformed IOD to Time Petri Net with energy constraints. It has performed an early analysis and validation of the embedded real-time systems' time and energy conditions. Tebibel's studies [3,6] have attributed formal semantics to IOD by a translation to (HCPNs). Besides the IOD control flow semantics formalization, they exploited CPN tools for the resulting HCPNs simulation and analysis. An attempt [24] has been made at formalizing the hierarchical use of IOD, by extending the work presented in [3] using timed CPNs. The approach presented in [2] has taken full advantage of the rich description in combining different interaction nodes into IODs. Some constructs of IODs have been formalized using a temporal logic called TRIO. Further, the authors have verified some user-defined properties by semantics' implementation in the Zot tool. The proposed approach in [11] has provided a Maude language formalization of UML IOD, with elementary interaction nodes. An automatic translation implementing the proposed formalization has been developed in the AToM3 tool [22]. Subsequently, the yielded Maude specification can be validated through simulation using the Maude system.

This paper proposes an automatic approach for formalizing UML IOD models, using the RL Maude language by integrating insights from two previous research works, namely [11] and [20]. The latter proposed a formalization of UML AD in the Maude language. It has treated the control nodes in UML ADs, which are chiefly similar to the overview level in UML IOD since an IOD is a variant of an AD. At the interaction level, our work revises and extends work [20] by introducing multiple complex interaction scenarios with nested combined fragments (CFs). Indeed, CFs model concurrent behaviors that complicate the analysis of the interaction nodes. Through this work, we intend to translate the IOD into a unified logical and semantic framework. That framework allows a rigorous and well-founded formal analysis to ensure the correctness and reliability of UML-based software systems.

Compared to other approaches, the primary advantage of our work lies in the versatility and universality of the Maude mathematical notations. Maude has good representational capabilities, which allow the integration of all concepts and notions of a language in a single semantic framework. Indeed, Maude does not use any linguistic construction that warps and hides the specific characteristics of each language or domain. "Everything in the Maude specifications is a direct definition of the

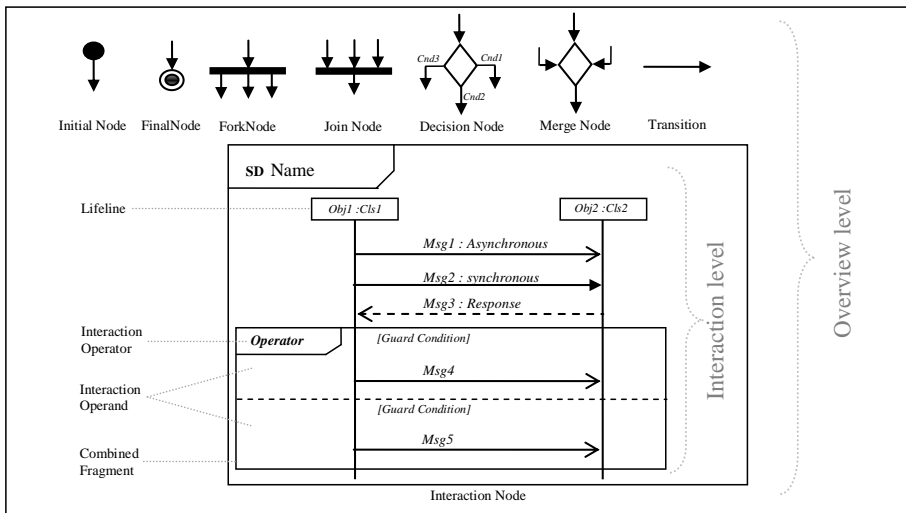
language” [13]. A Maude specification is executable. Therefore, Maude gives a formal executable specification to non-executable semi-formal UML models. Moreover, Maude boasts a highly extensible software infrastructure that functions as a mathematical environment. Within this framework, users can leverage a plethora of analysis techniques. Specifically simulation, model checking of invariants, and linear temporal logic (LTL) model checking, to effectively validate or verify properties inherent in Maude executable specifications.

### 3. Interaction overview diagram constructs

An IOD is a behavioral diagram in UML 2.0. It gives a complete, high-level abstract view of a system under development by showing the flow of interactions between its components. It is a two-level diagram that strikes a balance in providing a broad understanding of the system’s control flow and enabling in-depth exploration of specific interactions when required. It is adequate for yielding an overview of the flow of control, including synchronization, conditional branching, and activity concurrency. Further, it represents the interactions between different objects/actors in the system and pictures the overall dynamics of the system in a single diagram. Being a two-level diagram, an IOD helps designers manage the complexity of broad systems by decomposing the information into modular and understandable components. Thus, identifying potential issues early in the design process makes it easier to change the system to meet the requirements and save time and effort. An IOD shares the general structure of an AD to model the overview level of the system flow and describes interactions within the system using interaction diagram nodes that illustrate the invoked activities or operations.

Syntactically, see Figure 1, an IOD is a connected graph that uses UML AD control constructs (Initial, Decision, Merge, Fork, Join, and Final nodes) to illustrate control flow at the overview level. Instead of Activity elements, it uses rectangular elements to represent interaction nodes that display UML interaction diagrams. An interaction node can be any diagram of the four interaction diagram types (Sequence, Timing, Communication, or another interaction diagram). In this work, we are interested in IODs where interaction nodes are UML Sequence Diagrams (SDs). The utilization of SDs as interaction nodes simplifies the design process and focuses on a sequence of execution for different interactions in the system [29]. Interactions comprise, at the Interaction level, a set of lifelines. Each lifeline symbolizes a role attributed to an object/actor associated with a pertinent class in the system. Interactions over time are depicted as methodical, top-down-arranged messages, visually represented by arrows leading from the source lifeline to the target lifeline

Further, interactions can be regrouped compactly and concisely with Combined Fragments (CFs). CFs (loop, alternative, Option, etc.) define multiple types of control flows using an operator and one or more interaction operands.



**Figure 1.** Interaction Overview Diagrams Constructs

The interaction operator shows the logic of the fragment. It describes the semantics of the CF and determines how to use its interaction operands. An interaction operand is a container that groups the interaction fragments and messages exchanged, assuming the guard condition(s). Table 1 provides information on some types of operators and their corresponding descriptions.

**Table 1**  
Some Combined Fragments Operators

Operators	Description
Alt	Alternative: It represents a choice of behavior (among several operands)
Opt	Option: It represents the choice of behavior. It has only one operand to be selected if the guard condition is evaluated to true. Otherwise, the execution flow skips the behavior
Break	It represents a breaking scenario. It has one operand with guard condition. If the guard is true, the operand is chosen, and the rest of the enclosing interaction is skipped. Otherwise, the Break operand is ignored, and the rest of the enclosing interaction is selected
Par	Parallel: It represents the parallel execution of behaviors described within its operands
Loop	It represents an iterated execution of behavior a defined number of times or when a guard condition is evaluated as false
Strict	Strict Sequencing: It defines the strict ordering of the operands
Neg	Negative: It represents a set of negative traces that occur when the system has failed

## 4. Rewriting logic & Maude language

This section presents Rewriting Logic (RL) and the underlying Maude language. Since its introduction as a unifying framework for concurrency, RL has proven its completeness and suitability as an ideal logic for the specification and analysis of concurrent systems, where concurrent computation is precisely represented by logical deduction [27]. It is an executable semantic framework where different computational models can be specified and executed. More precisely, it is excellent support for giving precise semantics to a variety of concurrent model notations by assigning fully defined formal executable specifications.

A specification in RL is named rewriting theory. Essentially, a rewriting theory includes a signature (which makes up an equational theory) and a set of labeled (conditional) rewriting rules. The signature characterizes the static structure of the system (data structures and operations on them), whereas rewriting rules model the dynamic of the system features [30]. Based on rewriting logic, Maude is a mathematically well-founded declarative specification and programming language where the basic units of specification and programming are theories in RL [28]. The Maude system supports algebraic specification execution and formal analysis technique application. A Maude program (called a Maude module) is precisely a rewriting theory satisfying elementary execution conditions. The calculation in Maude involves logical inference through rewriting. Maude's modules are user-definable and are of two types:

- The functional module defines the system's static aspect as an equational theory. It is a specification in membership equational logic, where data types are declared and operations that act upon them are defined in a precise and rigorous way.
- system module specifies the behavior of a system. It models the dynamics of concurrent systems as a set of rewriting rules that describe local transitions between states [8].

Besides its expressiveness, Maude provides formal reasoning capabilities, including automatic validation and verification tools. For system specification verification, Maude supports the model-checking technique. Nowadays, it is the most popular technique used to prove that a system has no faults and meets its specifications. Maude's model checker is a Linear Temporal Logic properties (LTL properties) verifier. Under the right conditions, the Maude model checker tool can verify LTL properties over a logical finite state space [17]. The result of the verification can either be:

- A fix point (a finite state space) is reached, and the formula is fully verified.
- Conversely, an actual counter-example is offered, proving the violation of the property in question.

## 5. Formalization of IOD diagrams using Maude language

This section showcases the UML IOD formalization using Rewriting Logic and Maude. By formalizing, we intend to permit the verification and validation of UML diagrams based on the analysis results obtained from the equivalent Maude specifications. To

formalize the UML IOD using Maude language, we have first defined a Basic.IOD functional module that represents the static aspect of UML IOD described using basic types and operations on them. This module is shown in Figure 2.

```
fmod BASIC-IOD is
including STRING .
sort IOD-Config .
op none : -> IOD-Config(ctor) .
op __ : IOD-Config IOD-Config -> IOD-Config(assoccomm id: none) .
**** sort of activity chart : the overview level
Sorts NodeName NodeType IntractionName Interaction .
ops Initial Final : ->NodeName .
ops InitialNode FinalNode : ->NodeType .
op<_> : NodeName NodeType -> IOD-Config .
op[_] : IntractionName Interaction -> IOD-Config .
ops Start End : -> Interaction .
**** Definition sort and operation of Interaction in SD Node
*** object definition
sorts Object Oid ObjType .
ops Actor Obj : ->ObjType .
op _ : Oid ObjType -> Object .
*** message definition
Sorts Msg MsgId MsgType .
Ops Syn Asyn Rep : ->MsgType .
op _ : MsgId MsgType ->Msg .
*** exchanged and sanning Message :an interaction between objects.
Sort MsgSending .
Subsort MsgSending<Interaction .
op<_> : Object Msg Object ->MsgSending .
*** Combined Fragment definition
Sorts CombFragment OperdName Condition .
Subsort CombFragment<Interaction .
subsort String <Condition
op Par{_:} : OperdName MsgSending ->CombFragment .
op Par{_:} : OperdName CombFragment ->CombFragment .
op Opt{_:} : Condition MsgSending ->CombFragment .
op Opt{_:} : Condition CombFragment ->CombFragment .
op Alt{_:} : Condition MsgSending ->CombFragment .
op Alt{_:} : Condition CombFragment ->CombFragment .

**** Loop definition
sort LoopStatus .
ops Entry Exist : ->LoopStatus .
op Loop{_:} : Nat Nat Condition LoopStatus ->CombFragment .
op Loop{_:} : Nat Nat Condition MsgSending ->CombFragment .
op Loop{_:} : Nat Nat Condition CombFragment ->CombFragment .

endfm
```

**Figure 2.** Basic Interaction Overview Diagram Functional Module



In the Basic.IOD module, we define a new type called ‘IOD\_Config’ that represents the current nodes of an IOD diagram instance (execution occurrence). The ‘none’ constant denotes the empty configuration in an IOD. In our approach, current nodes are the Initial node, Final node or interaction node. To specify the initial node and final node, we define the operation “< \_ : \_ >”. The first parameter of this operation is a constant of the ‘NodeName’ sort which represents the name of the node, whereas the second one is used to specify the type of the node, which can be ‘initial Node’ or ‘final Node’.

The interaction nodes in IOD are defined by the operation “[\_ ]”, where the first parameter of this operation is the name of the interaction node, whilst the second one is used to represent an interaction inside the node (message passing between objects). We have also defined in the second parameter two flag values (constants) denoting the beginning of the invocation (“Start”) and the end of the invocation (“End”) of the interaction node.

The control flow between interaction nodes in the overview level is formalized using rewriting rules. Rewriting rules represent transitions firing or controlling nodes with their conditions. We note the proposed overview level formalization is adapted from [20] that formalizes UML AD using Maude language. Table 2 summarizes the rewriting rules corresponding to the principal structures of the overview level.

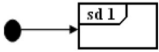
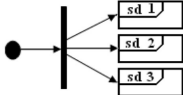
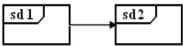

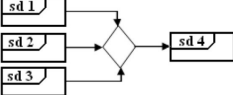
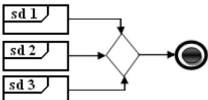
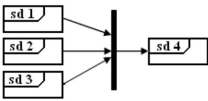
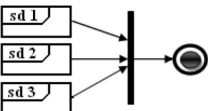
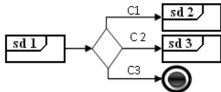
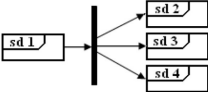
Interaction nodes in the interaction level of the IOD emphasize object interactions. An interaction node contains lifelines (object/Actor) and exchanged messages to represent a single scenario as a smaller SD.

To describe the sending of messages between objects in Maude, we have created two general sorts, called (“Object” and “Msg”). The first one is defined using the operation “\_:\_” where the first parameter is a constant of the “Oid” sort that represents the object name, whereas the second one is used to specify the object type. The “Msg” sort displays a message defined by the operation “\_:\_” that denotes the “id” of the message in the first parameter and the type in the former. An exchanged message (a simple interaction) is outlined with the operation “< \_, \_, \_ >” (defined in “MsgSending” sort) where the first parameter is the sender object, the second is the message sent, and the last one is the receiver object. Since sent messages are attached to the name of the interaction node where they participate, “MsgSending” types are declared as sub-sorts of the “Interaction” sort, which is the current interaction.

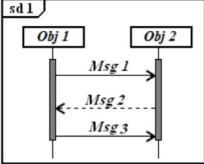
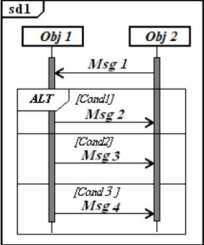
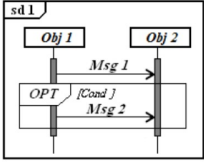
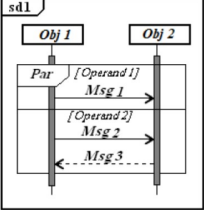
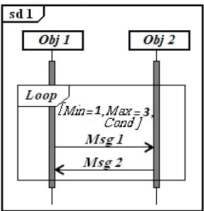
Furthermore, interactions in IOD diagram can be represented in a compacted form using Combined Fragments (CFs). A CF is defined by an operator which specifies how the operands will be executed. In our approach, we define for each CF operator an operation in Maude (as shown in Basic.IOD module in Figure 2). Table 3 depicts the rewriting rules corresponding to each operator.

We note that these rules are valid even for interactions between three objects and more, and the proposed formalization allows the nesting of the CFs.

**Table 2**  
Mapping Control Structures to Maude

Interaction Overview Diagram Overview Level	Corresponding Maude Rewriting Rules
<p>Initial to SD</p> 	$rl \text{ [Initial]} : < \text{Initial} : \text{InitialNode} > => [\text{SD1}   \text{Start}] .$
<p>Initial to Fork Node</p> 	$rl \text{ [Initial2Frk]} : < \text{Initial} : \text{Initial Node} > => [\text{SD1}   \text{Start}] [\text{SD2}   \text{Start}] [\text{SD3}   \text{Start}] .$
<p>SD to SD</p> 	$rl \text{ [transition]} : [\text{SD1}   \text{End}] => [\text{SD2}   \text{Start}] .$
<p>SD to Final Node</p> 	$rl \text{ [ToFinal]} : [\text{SD1}   \text{End}] => < \text{Final} : \text{FinalNode} > .$
<p>Merge Node</p> 	$rl \text{ [Merge]} : [\text{SD1}   \text{End}] => [\text{SD4}   \text{Start}] .$ $rl \text{ [Merge]} : [\text{SD2}   \text{End}] => [\text{SD4}   \text{Start}] .$ $rl \text{ [Merge]} : [\text{SD3}   \text{End}] => [\text{SD4}   \text{Start}] .$
	$rl \text{ [Merge]} : [\text{SD1}   \text{End}] => < \text{Final} : \text{FinalNode} > .$ $rl \text{ [Merge]} : [\text{SD2}   \text{End}] => < \text{Final} : \text{FinalNode} > .$ $rl \text{ [Merge]} : [\text{SD3}   \text{End}] => < \text{Final} : \text{FinalNode} > .$
<p>Join Node</p> 	$rl \text{ [Join]} : [\text{SD1}   \text{End}] [\text{SD2}   \text{End}] [\text{SD3}   \text{End}] => [\text{SD4}   \text{Start}] .$
	$rl \text{ [Join]} : [\text{SD1}   \text{End}] [\text{SD2}   \text{End}] [\text{SD3}   \text{End}] => < \text{Final} : \text{FinalNode} > .$
<p>Decision Node</p> 	$rl \text{ [Decision\_C1]} : [\text{SD1}   \text{End} \{ \text{Cond} \}] => [\text{SD2}   \text{Start}] .$ $rl \text{ [Decision\_C2]} : [\text{SD1}   \text{End} \{ \text{Cond} \}] => [\text{SD3}   \text{Start}] .$ $rl \text{ [Decision\_C3]} : [\text{SD1}   \text{End} \{ \text{Cond} \}] => < \text{Final} : \text{FinalNode} > .$
<p>Fork Node</p> 	$rl \text{ [Fork]} : [\text{SD1}   \text{End}] => [\text{SD2}   \text{Start}] [\text{SD3}   \text{Start}] [\text{SD4}   \text{Start}] .$

**Table 3**  
Mapping Interactions to Maude

IOD lower level	Corresponding Maude Rewriting Rules
<p>Message Passing (Simple Message)</p> 	<pre> ri [SendMsg1] : [SD1   Start ] =&gt; [SD1   &lt; Obj1 : Obj, Msg1 : Syn, Obj2 : Obj&gt; ] . ri [SendMsg2] : [SD1   &lt; Obj1 : Obj, Msg1 : Syn, Obj2 : Obj&gt; ] =&gt; [SD1   &lt; Obj2 : Obj, Msg2 : Ans, Obj1 : Obj&gt; ] . ri [SendMsg3] : [SD1   &lt; Obj2 : Obj, Msg2 : Ans, Obj1 : Obj&gt; ] =&gt; [SD1   &lt; Obj1 : Obj, Msg3 : Asy, Obj2 : Obj&gt; ] . ri [endSD] : [SD1   &lt; Obj1 : Obj, Msg3 : Asy, Obj2 : Obj&gt; ] =&gt; [SD1   End ] . </pre>
<p>Alt Fragment</p> 	<pre> ri [SendMsg1] : [SD1   Start ] =&gt; [SD1   &lt; Obj2 : Obj, Msg1 : Asy, Obj1 : Obj&gt; ] . *** Alt Condition 1 is true ri [AltCond1] : [SD1   &lt; Obj2 : Obj, Msg1 : Asy, Obj1 : Obj&gt; ] =&gt; [SD1   Alt {Cond1 : &lt; Obj1 : Obj, Msg2 : Asy, Obj2 : Obj&gt; } ] . ri [AltCond1] : [SD1   Alt {Cond1 : &lt; Obj1 : Obj, Msg2 : Asy, Obj2 : Obj&gt; } ] =&gt; [SD1   End ] . *** Alt Condition 2 is true ri [AltCond2] : [SD1   &lt; Obj2 : Obj, Msg1 : Asy, Obj1 : Obj&gt; ] =&gt; [SD1   Alt {Cond2 : &lt; Obj1 : Obj, Msg3 : Asy, Obj2 : Obj&gt; } ] . ri [AltCond2] : [SD1   Alt {Cond2 : &lt; Obj1 : Obj, Msg3 : Asy, Obj2 : Obj&gt; } ] =&gt; [SD1   End ] . *** Alt Condition 3 is true or else statement ri [AltCond3] : [SD1   &lt; Obj2 : Obj, Msg1 : Asy, Obj1 : Obj&gt; ] =&gt; [SD1   Alt {Cond3 : &lt; Obj1 : Obj, Msg3 : Asy, Obj2 : Obj&gt; } ] . ri [AltCond3] : [SD1   &lt; Obj1 : Obj, Msg3 : Asy, Obj2 : Obj&gt; ] =&gt; [SD1   End ] . </pre>
<p>Opt Fragment</p> 	<pre> ri [SendMsg1] : [SD1   Start ] =&gt; [SD1   &lt; Obj1 : Obj, Msg1 : Asy, Obj2 : Obj&gt; ] . *** Opt Condition is True ri [CondOptTrue] : [SD1   &lt; Obj1 : Obj, Msg1 : Asy, Obj2 : Obj&gt; ] =&gt; [SD1   Opt {Cond : &lt; Obj1 : Obj, Msg2 : Asy, Obj2 : Obj&gt; } ] . ri [.....] : [SD1   Opt {Cond : &lt; Obj1 : Obj, Msg2 : Asy, Obj2 : Obj&gt; } ] =&gt; [SD1   End ] . *** Opt Condition is false ri [CondOptFalse] : [SD1   &lt; Obj1 : Obj, Msg1 : Syn, Obj2 : Obj&gt; ] =&gt; [SD1   End ] . </pre>
<p>Par Fragment</p> 	<pre> ri [ParFragment] : [SD1   Start ] =&gt; [SD1   Par{Operand1 : &lt; Obj1 : Obj, Msg1 : Asy, Obj2 : Obj&gt; } ] . [SD1   Par{Operand2 : &lt; Obj1 : Obj, Msg2 : Syn, Obj2 : Obj&gt; } ] .  ri [Par_Msg2] : [SD1   Par{Operand2 : &lt; Obj1 : Obj, Msg2 : Syn, Obj2 : Obj&gt; } ] =&gt; [SD1   Par{Operand2 : &lt; Obj2 : Obj, Msg3 : Ans, Obj1 : Obj&gt; } ] .  ri [EndPar] : [SD1   Par{Operand1 : &lt; Obj1 : Obj, Msg1 : Asy, Obj2 : Obj&gt; } ] [SD1   Par{Operand2 : &lt; Obj2 : Obj, Msg3 : Ans, Obj1 : Obj&gt; } ] =&gt; [SD1   End ] . </pre>
<p>Loop Fragment</p> 	<pre> vars count max : Nat . varcond : String . ri [Start Loop] : [SD1   Start ] =&gt; [SD1   Loop{1,3,"Cond" : Entry} ] . *** Loop execution cr1 [LoopMsg1] : [SD1   Loop{ count, max, cond : Entry} ] =&gt; [SD1   Loop{ count, max, cond : &lt; Obj1 : Obj, Msg1 : Asy, Obj2 : Obj&gt; } ] if ( cond == "Cond" ) /\ ( count &lt;= max ) . ri [LoopMsg2] : [SD1   Loop{ count, max, cond : &lt; Obj1 : Obj, Msg1 : Asy, Obj2 : Obj&gt; } ] =&gt; [SD1   Loop{ count, max, cond : &lt; Obj2 : Obj, Msg2 : Asy, Obj1 : Obj&gt; } ] . *** Loop restart(with "Cond" OR "NotCond") ri [RestartLoopCond] : [SD1   Loop{ count , max, cond : &lt; Obj2 : Obj, Msg2 : Asy, Obj1 : Obj&gt; } ] =&gt; [SD1   Loop{ count +1, max, "Cond" : Entry } ] . ri [RestartLoopNotCond] : [SD1   Loop{ count , max, cond : &lt; Obj2 : Obj, Msg2 : Asy, Obj1 : Obj&gt; } ] =&gt; [SD1   Loop{ count +1, max, "NotCond" : Entry } ] . **** Loop exist cr1 [ExistWithoutLoop] : [SD1   Loop{ count, max, cond : Entry} ] =&gt; [SD1   Loop{ count , max, cond : Exist } ] if ( cond /= "Cond" ) or ( count &gt; max ) . ri [EndLoop] : [SD1   Loop{ count, max, cond : Exist } ] =&gt; [SD1   End ] . </pre>

## 6. The MDE based approach

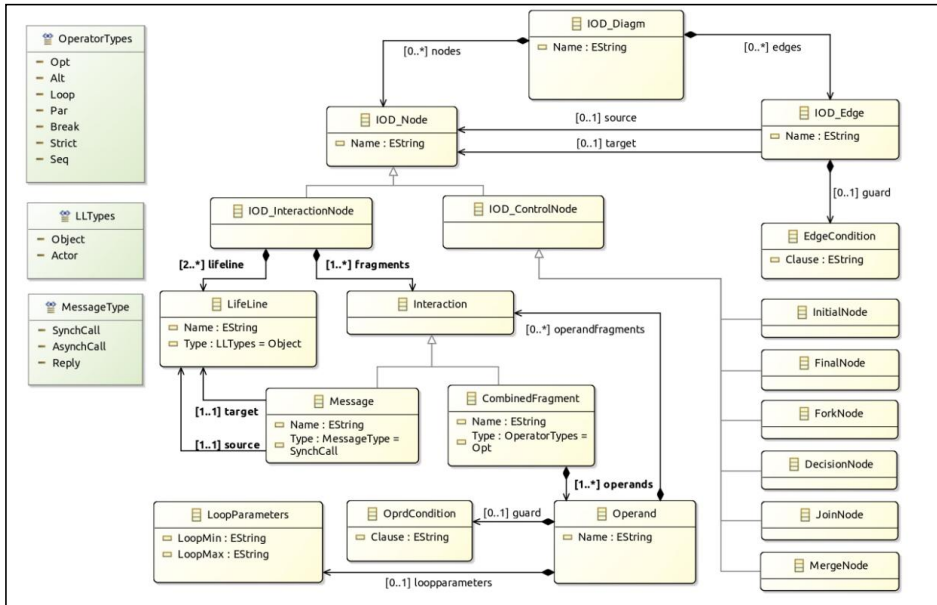
In this section, we outline the proposed MDE-based approach to transform the dynamic behaviors of systems expressed using UML IODs into their equivalent Maude specification, by considering the defined formalization in Section 5. The approach comprises a two-step process:

- The first step is to redefine a simplified meta-model for the IOD and build a graphical editor for the diagram according to the proposed meta-model. To fulfill this, we have offered the use of Eclipse Modeling Framework (EMF) [15], which forms the basis for MDE on the Eclipse platform. Based on the proposed IOD meta-model, we have used Sirius framework [16] to build a graphical modeling editor for IOD.
- The second step is to prepare the generation of the equivalent Maude specification of the IOD. For achieving an automatic and correct process of code generation, we have proposed to use the Acceleo language [14] to define and implement the transformations.

In the ensuing section, we unveil in detail our approach.

### 6.1. IOD Meta-Model

The development of modeling language requires providing both abstract syntax and concrete syntax. Abstract syntax specifies the meta-model constructs, the associated attributes, relationships, and constraints. The concrete syntax determines how the previous constructs are connected and how they visually appear in a graphical editor. In this work, we deal with the formalization and model analysis of a subset of the UML 2.5 meta-model concepts by using a simplified meta-model of the diagram. In Eclipse EMF, a meta-model is established in the Ecore format [33], which is primarily a subset of UML class diagrams. We propose to meta-model the UML IOD with the Ecore model shown in Figure 3. The root node of this meta-model (named `IOD_Diagram`) contains several nodes, namely (`IOD_Nodes`), each modeling either a control node (`IOD_ControlNode`) or an interaction node (`IOD_InteractionNode`). These nodes are connected by edges (`IOD_edges`). These edges can have guards to conditionally branch the control control. There are several types of control nodes for determining the flow direction. This follows the notion of UML AD. `InitialNode`, `FinalNode`, `DecisionNode`, `ForkNode`, `JoinNode` and `MergeNodes` classes describe the control flow at the overview level in the IOD. The interaction nodes are based on the UML SD concepts. An interaction node consists of several interactions among objects/actors. The lifelines (`LifeLine` class) display the interacting objects/actors by exchanging messages (`Message` class). CFs (`CombinedFragment` class) are a composition of interactions defined by an interaction operator (which is specified through the `Type` attribute), and embedded in corresponding interaction operands (the `Operator` class). An operand can have interaction constraints (such as guards or loop parameters for a loop operator).



**Figure 3.** Interaction Overview Diagram Meta-Model

Using the proposed Ecore model, we can generate a user-friendly tree-based editor for the UML IOD modeling language. This editor allows easy editing and viewing of model instances through EMF. To develop its graphical modeling editors, we use the Sirius framework. Eclipse, EMF, and Sirius technology grant the creation of customized graphical modeling environments. The users can create, visualize, and update IOD models by using the custom modeling environment. (see the palette toolbar in Figure 5).

## 6.2. Maude Code generation

The next step is to transform the UML IOD specification into its equivalent Maude specification. According to the presented formalization in Section 5. The transformation is performed by using the Acceleo transformation language. Acceleo is a template-based technology. It includes tools for creating custom code generators that allow the automatic generation of any code from a data source available in an EMF format. Acceleo models faithfully translate the IOD formalization into corresponding constructs in the Maude language. To establish Maude's specification, we have defined a set of atomic mapping rules from the elementary elements and structures in the source model into elementary constructs in the target model (as defined in Section 5: Figure 2, Table 2, and Table 3). The Maude code generation process is composed of a set of Acceleo templates. Figure 4 shows some templates.

```

[template public IOD2Maude (anIOD_Diagm : IOD_Diagm)]
[comment @main/]
-----Maude code generation of the overview level
[file (anIOD_Diagm.Name.concat('.maude'),false,'UTF-8')]
mod [anIOD_Diagm.Name/] is
including BASIC-IOD .
[for (aIOD_InteractionNode :IOD_InteractionNode|anIOD_Diagm.eContents(IOD_InteractionNode))]
including [aIOD_InteractionNode.Name/] .
[/for]
Op [anIOD_Diagm.Name/] :-> IOD-Config .
Eq [anIOD_Diagm.Name/] = <Initial:InitialNode> .
--- Maude code generation of control flow between interaction nodes
[anIOD_Diagm.GenControlFlow()]
endm
[/file]
-----Maude code generation of the interaction nodes level
[for (anIOD_InteractionNode :IOD_InteractionNode|anIOD_Diagm.eContents(IOD_InteractionNode))]
[file (anIOD_InteractionNode.Name.concat('.maude'),false,'UTF-8')]
Mod [anIOD_InteractionNode.Name/] is
including BASIC-IOD .
---The objects/actors definition
[anIOD_InteractionNode.GenObject()]
---The messages definition
[anIOD_InteractionNode.GenMessage()]
---The CF definition
[anIOD_InteractionNode.GenCombinedFragment ()/]
--- Maude code generation of the interactions: messages exchange
op[anIOD_InteractionNode.Name/] :-> IntractionName.
[anIOD_InteractionNode.GenInteractions()/]
endm
[/file]
[/for]
[/template]

[comment : GenControlFlow template: generation of control flow in Maude /]
[template private GenControlFlow (anIOD_Diagm :IOD_Diagm)]
--- code Maude generation of sequential transition
[for (aIOEdge :IOD_Edge|anIOD_Diagm.eAllContents(IOD_Edge)
->select(a|[a.source.ocIsKindOf(InitialNode) and a.target.ocIsKindOf(IOD_InteractionNode)]
or [a.source.ocIsKindOf(IOD_InteractionNode) and a.target.ocIsKindOf(IOD_InteractionNode)]
or [a.source.ocIsKindOf(IOD_InteractionNode) and a.target.ocIsKindOf(FinalNode)]))]
[if aIOEdge.source.ocIsKindOf(InitialNode) and aIOEdge.target.ocIsKindOf(IOD_InteractionNode)]
r1 ['/'/Initial]:<Initial:InitialNode>=>['/'/[aIOEdge.target.Name/]|Start].
[elseif aIOEdge.source.ocIsKindOf(IOD_InteractionNode) and aIOEdge.target.ocIsKindOf(IOD_InteractionNode)]
r1 ['/'/[aIOEdge.source.Name/]|'-To-
/'/[aIOEdge.target.Name/]|:]<['/'/[aIOEdge.source.Name/]|End]=>['/'/[aIOEdge.target.Name/]|Start].
[elseif aIOEdge.source.ocIsKindOf(IOD_InteractionNode) and aIOEdge.target.ocIsKindOf(FinalNode) ]
r1 ['/'/ToFinal]:['/'/[aIOEdge.source.Name/]|End]=><Final:FinalNode>.
[/if]
[/for]
--- code Maude generation of control nodes
[for (aIOD_ControlNode :IOD_ControlNode|anIOD_Diagm.eAllContents(IOD_ControlNode)
->select(a|not(a.ocIsKindOf(InitialNode) or a.ocIsKindOf(FinalNode)))]
[if aIOD_ControlNode.ocIsKindOf(ForkNode)] [aIOD_ControlNode.GenForkNode()/]
[elseif aIOD_ControlNode.ocIsKindOf(MergeNode)] [aIOD_ControlNode.GenMergeNode()/]
[elseif aIOD_ControlNode.ocIsKindOf(DecisionNode)] [aIOD_ControlNode.GenDecisionNode()/]
[elseif aIOD_ControlNode.ocIsKindOf(JoinNode)] [aIOD_ControlNode.GenJoinNode()/]
[/if]
[/for]
[/template]

[comment : GenObject template: generation of Objects/actors definitions in Maude /]
[template private GenObject(anIOD_InteractionNode : IOD_InteractionNode) post(tokenize("\n"))]
[if anIOD_InteractionNode.lifeline<=null]
ops
[for (aName : String|anIOD_InteractionNode.lifeline.Name->asSet())][aName/] [/for] :-> Oid .
[/if]
[/template]

```

Figure 4. Some Aceleo templates to generate Maude code

The templates browse the source model elements (instance of the UML IOD metamodel) and generate the corresponding Maude code. The main *Acceleo* template is named *IOD2Maude()*. It generates the equivalent Maude code for both levels of the diagram: the overview level and the interaction level. For the overview level, it generates a source Maude code file for the statements of the corresponding Maude modules for each interaction node. After that, the *GenControlFlow()* template generates the Maude code for the control flow between the interaction nodes. The *GenControlFlow()* template first generates the equivalent rewriting rule for each sequential transition firing. Afterward, according to the defined semantics, the appropriate template is used to generate the corresponding rewriting rule(s) for each type of control node. The main template generates a text file for each interaction node at the interaction level. The text file saves the resulting Maude code from the mapping of the underlined interaction objects, messages, CFs and interactions by applying different templates. For example, the *GenObjet()* template is applied for the generation of the node-underlined object/actor Maude specification.

## 7. Case study

The efficiency of our approach is assessed through an online virtual bookstore [35] example. A virtual bookstore is a digital framework that allows customers to purchase books online. The bookstore typically has a wide selection of accessible books for sale, which can be easily searched and filtered based on various criteria. Customers can access the online bookstore through a front-end-user interface. They can enter keywords to search for specific titles or authors, browse the books listed in the library, view book details, and order books. To order books, customers can add books to their shopping cart, view their orders, and make secure online payments. Figure 5 shows an IOD that depicts the behavioral aspect of the online bookstore system. The IOD portrays essential nodes, namely: initial, final, decision, merge, and four interaction nodes (*OrderBooks*, *ConnectToSystem*, *RegisterOrder*, and *OrderPayment*). Each interaction node includes a UML SD that involves three actors: the customer, the lib interface (user interface), and the lib system control. The latter monitors the overall operations of the online bookstore system. It carries out operations, including inventory management and payment processing for books.

The diagram starts at the initial node and leads to the first interaction node that models the book order. In this node, the system iterates over each keyword in the search query and retrieves a list of matching books. Once the system has retrieved a list of books for each keyword in the search query, it creates a new list of books that fits all the keywords in the query. The last list is added to a search results set and displayed to the user. Customers can cancel the order at any point before completing the checkout process. In this scenario, the diagram concludes at the final node. Before finalizing the order, the online bookstore invites the customer to log into his account if he is not already logged in. For a new customer, the online bookstore asks him to create a new account. Once a customer has connected to his account, he can complete

the order. He will confirm the details of his order in the RegisterOrder interaction node by entering his shipping and billing information. Then, in the OrderPayment interaction node, he makes the payment using a secure online payment system. The OrderPayment interaction node introduces a new actor named CreditCardOperator. This actor manages the responsibility of transaction payments between the bookstore and the customer.

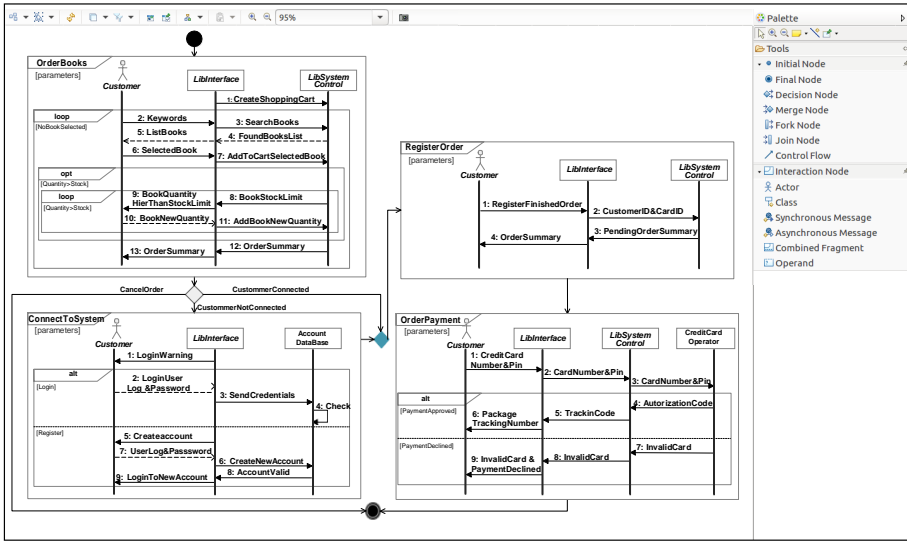


Figure 5. Specification of the OnlineBookStore IOD

## 7.1. Generation of Maude specification

To thoroughly assess the virtual bookstore model, it is imperative to map it into its equivalent Maude specification. The bookstore model Maude formalization follows patterns and instructions specified in the Aceleo templates. Aceleo templates define how to navigate the elements of the bookstore as the EMF input model and what code to generate. After the Aceleo template execution, the corresponding Maude code to the model of the virtual bookstore is generated. We have used Maude's modularity feature to generate separate Maude modules for the overview level and each interaction node. Figure 6 shows the Maude system module of the OnlineBookStore IOD. It outlines the Maude specification of the virtual bookstore model.

This module includes the Basic.IOD functional module, where the static aspect of the diagram (actors, messages, CFs, interactions) is declared using different types and operators. Each interaction node's code is also included by calling its corresponding Maude module. In addition, this module defines the operator OnlineBookStore as IOD-Config sort. It corresponds to the name of the IOD model. It also introduces the OnlineBookStore equation to indicate the first state of the IOD's execution. The



module ends with the equivalent rewriting rule(s) for each sequential transition firing or control node.

```

mod OnlineBookStore is
  including BASIC-IOD .
  including OrderBooks .
  including ConnectToSystem .
  including RegisterOrder .
  including OrderPayment .
  op OnlineBookStore : -> IOD-Config .
  eq OnlineBookStore = < Initial : InitialNode > .
  --- The control flow between interactions
  rl [Initial] : < Initial : InitialNode > => [ OrderBooks | Start ] .
  rl [Decision-CancelOrder] : [ OrderBooks | End ] => < Final : FinalNode > .
  rl [Decision-CustomerNotConnected] : [ OrderBooks | End ] => [ ConnectToSystem | Start ] .
  rl [Decision-CustomerConnected] : [ OrderBooks | End ] => [ RegisterOrder | Start ] .
  rl [ConnectToSystem-To-RegisterOrder] : [ConnectToSystem | End ] => [ RegisterOrder | Start ] .
  rl [RegisterOrder-To-OrderPayment] : [RegisterOrder | End ] => [OrderPayment | Start ] .
  rl [ToFinal] : [OrderPayment | End ] => < Final : FinalNode > .
endm

```

**Figure 6.** The Generated OnlineBookStore System Module: the overview level of IOD

Each interaction node is depicted in a different system module. For example, The RegisterOrder system module, shown in Figure 7, contains the rewriting rules specifying interactions (exchanging messages) between different objects in the RegisterOrder interaction node.

```

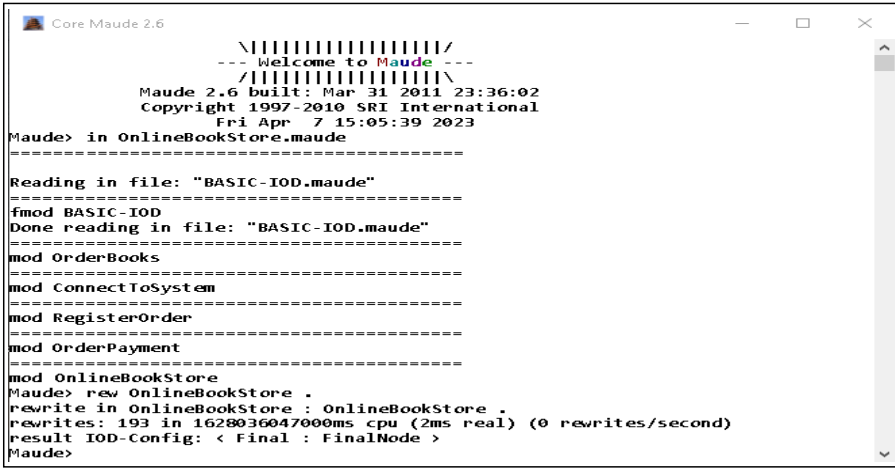
mod RegisterOrder is
  including BASIC-IOD .
  --- The objects
  ops Customer LibInterface LibSystemControl : -> Oid .
  --- The OrderSummary messages
  ops RegisterFinishedOrder CustomerID&CardID PendingOrderSummary OrderSummary : -> MsgId .
  --- The interaction
  op RegisterOrder : -> IntractionName .
  rl [RegisterFinishedOrder] : [ RegisterOrder | Start ]
    => [ RegisterOrder | < Customer : Skm , RegisterFinishedOrder : Syn , LibInterface : Obj > ] .
  rl [CustomerID&CardID] : [ RegisterOrder | < Customer : Skm , RegisterFinishedOrder : Syn , LibInterface : Obj > ]
    => [ RegisterOrder | < LibInterface : Obj , CustomerID&CardID : Syn , LibSystemControl : Obj > ] .
  rl [PendingOrderSummary] : [ RegisterOrder | < LibInterface : Obj , CustomerID&CardID : Syn , LibSystemControl : Obj > ]
    => [ RegisterOrder | < LibSystemControl : Obj , PendingOrderSummary : Syn , LibInterface : Obj > ] .
  rl [OrderSummary] : [ RegisterOrder | < LibSystemControl : Obj , PendingOrderSummary : Syn , LibInterface : Obj > ]
    => [ RegisterOrder | < LibInterface : Obj , OrderSummary : Syn , Customer : Skm > ] .
  rl [RegisterOrderEnd] : [ RegisterOrder | < LibInterface : Obj , OrderSummary : Syn , Customer : Skm > ]
    => [ RegisterOrder | End ] .
endm

```

**Figure 7.** The Generated RegisterOrder System Module  
from the RegisterOrder Interaction Node

## 7.2. Simulation

We invoked the RL Maude system to perform the resulting Maude specification analysis by simulation. The simulation process involves evolving the system to observe its behavior. In the Maude system, the simulation is composed of iterations where the initial state of the diagram changes by applying one or more rewriting rules. Besides the generated file, the user can limit the number of rewriting steps in the simulator if (s/he) wants to check intermediate states. Otherwise, the simulator continues the simulation operation until reaching a final state for a correct design. The results of the simulation of the library Maude specification are presented in Figure 8. We notice that when the simulator starts from the initial node, the final node is reached.



```

Core Maude 2.6
-----
Welcome to Maude
-----
Maude 2.6 built: Mar 31 2011 23:36:02
Copyright 1997-2010 SRI International
Fri Apr 7 15:05:39 2023

Maude> in OnlineBookStore.maude
=====
Reading in file: "BASIC-IOD.maude"
=====
fmod BASIC-IOD
Done reading in file: "BASIC-IOD.maude"
=====
mod OrderBooks
=====
mod ConnectToSystem
=====
mod RegisterOrder
=====
mod OrderPayment
=====
mod OnlineBookStore
Maude> rew OnlineBookStore .
rewrite in OnlineBookStore : OnlineBookStore .
rewrites: 193 in 1628036047000ms cpu (2ms real) (0 rewrites/second)
result IOD-Config: < Final : FinalNode >
Maude>

```

Figure 8. The Simulation of the Virtual Bookstore IOD within Maude System

## 7.3. Verification and analysis

This section illustrates the use of the Maude LTL model checker to perform the verification and analyses. LTL is a formalism for expressing system properties using predicates. As Maude supports equational logic, the LTL properties are expressed using equations. To verify a property, we need to define it manually. After its definition, the property is embedded in the specification module, where the system and its behavior are defined. Consider the virtual bookstore specification in the example. A temporal property that the specification must satisfy is termination. The property means that when the system starts from the initial node, it always reaches the final node. We define two predicates (Start and Final) to express the termination property in a new module called IODPREDS. The IODPREDS module encompasses the property along with the bookstore system specification (see Figure 9). The termination property is expressed in LTL as follows:

$$\Box \langle \rangle \text{Final}(\langle \text{Final} : \text{FinalNode} \rangle)$$

```

mod IOD-PREDS is
  protecting OnlineBookStore.
  including SATISFACTION .
  subsort IOD-Config < State .
  op Start : IOD-Config -> Prop .
  op Final : IOD-Config -> Prop .
  var config : IOD-Config .
  eq < Initial : InitialNode > config |= Start (< Initial : InitialNode >) = true .
  eq < Final : FinalNode > config |= Final (< Final : FinalNode >) = true .
endm

```

Figure 9. IOD-PREDS Module


The IODCHECK module verifies this propriety (see Figure 10). In Figure 11, Maude's LTL model checker result shows that the property is verified successfully for the diagram.

```

mod IOD-CHECK is
  protecting IOD-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
endm

```

Figure 10. Check the LTL Termination Property



```

Core Maude 2.6
-----
Welcome to Maude
-----
Maude 2.6 built: Mar 31 2011 23:36:02
Copyright 1997-2010 SRI International
Fri Apr 7 15:33:10 2023

Maude> in IOD-CHECK.maude
Reading in file: "IOD-PREDS.maude"
Reading in file: "model-checker.maude"
=====
fmod LTL
fmod LTL-SIMPLIFIER
fmod SAT-SOLVER
fmod SATISFACTION
fmod MODEL-CHECKER
Done reading in file: "model-checker.maude"
Reading in file: "OnlineBookStore.maude"
=====
Reading in file: "BASIC-IOD.maude"
=====
fmod BASIC-IOD
Done reading in file: "BASIC-IOD.maude"
mod OrderBooks
mod ConnectIoSystem
mod RegisterOrder
mod OrderPayment
mod OnlineBookStore
Done reading in file: "OnlineBookStore.maude"
=====
mod IOD-PREDS
Done reading in file: "IOD-PREDS.maude"
=====
mod IOD-CHECK
=====
reduce in IOD-CHECK : modelCheck(OnlineBookStore, []<> Final(< Final : FinalNode >)) .
rewrites: 400 in 16346683478ms cpu (38ms real) (0 rewrites/second)
result Bool: true
Maude> _

```

Figure 11. LTL Termination Property Result

## 8. Conclusion

In this work, we have proposed a formal and an MDE-based approach to tackle UML IOD formalization and analysis using the Maude language. We have defined a subset of aspects of IODs using the EMF framework in the Eclipse environment. Further, we have used the Sirius framework to develop a visual modeling tool for editing and manipulating IOD models. Acceleo is a template-based framework for model transformation and code generation technology. Specifying the IOD early in the development cycle can help identify issues and gaps in system requirements. The choice of the RL and Maude language made the analysis techniques and verification tools accessible. We have shown how using Maude allows for simulation and execution analysis. We have also exploited the underlying LTL model checker to verify the diagram's correctness. In our case, we have checked the termination property using LTL formulas. The performance of the proposed approach has been evaluated through a virtual bookstore example. In this paper, we have mainly addressed the formalization of SD-type interaction nodes. In future works, we plan to embrace the proposed work for complex concurrent control flow paths and different types of interaction diagrams in IOD interaction nodes. We also plan to check other properties using Maude model checking and annotate the analysis results in the UML IOD diagram.

## 9. Tool and Acceleo templates

The program code and Acceleo templates in this work are publicly available through the GitHub repository<sup>1</sup>, necessary to run and execute for interpreting, replicating, and building on the findings reported in the paper.

## Acknowledgements

*The authors thank the reviewer(s) for their insightful comments and suggestions. The authors are also grateful to the Editor-in-Chief, the Editor, and the Editorial Office Assistant(s) for managing this manuscript.*

## References

- [1] Andrade E., Maciel P., Callou G., Nogueira B.: Mapping UML Interaction Overview Diagram to Time Petri Net for Analysis and Verification of Embedded Real-Time Systems with Energy Constraints. In: *2008 International Conference on Computational Intelligence for Modelling Control & Automation*, pp. 615–620, IEEE, 2008. doi: 10.1109/cimca.2008.44.
- [2] Baresi L., Morzenti A., Motta A., Rossi M.: From Interaction Overview Diagrams to Temporal Logic. In: *Models in Software Engineering. Workshops and Symposia at MoDELS 2010, Oslo, Norway, October 3–8, 2010, Reports and Revised Selected Papers*, pp. 90–104, Springer, 2010. doi: 10.1007/978-3-642-21210-9\_9.

---

<sup>1</sup><https://github.com/IODFormalization/IOD.TO-MAUDE>

- [3] Bennama M., Bouabana-Tebibel T.: Validation environment of UML2 IOD based on hierarchical coloured Petri nets, *International Journal of Computer Applications in Technology*, vol. 47(2-3), pp. 227–240, 2013. doi: 10.1504/ijcat.2013.054372.
- [4] Bernardi S., Donatelli S., Merseguer J.: From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. In: *WOSP'02: Proceedings of the 3rd International Workshop on Software and Performance*, pp. 35–45, 2002. doi: 10.1145/584369.584376.
- [5] Bernardi S., Merseguer J.: Performance evaluation of UML design with Stochastic Well-formed Nets, *Journal of Systems and Software*, vol. 80(11), pp. 1843–1865, 2007. doi: 10.1016/j.jss.2007.02.029.
- [6] Bouabana-Tebibel T.: Semantics of the interaction overview diagram. In: *2009 IEEE International Conference on Information Reuse & Integration*, pp. 278–283, IEEE, 2009. doi: 10.1109/iri.2009.5211565.
- [7] Bowen J.P., He J.: An algebraic approach to hardware compilation, *Modern Formal Methods and Applications*, pp. 151–176, 2006.
- [8] Bruni R., Meseguer J.: Semantic foundations for generalized rewrite theories, *Theoretical Computer Science*, vol. 360(1-3), pp. 386–414, 2006. doi: 10.1016/j.tcs.2006.04.012.
- [9] Clavel M., Durán F., Eker S., Escobar S., Lincoln P., Martí-Oliet N., Meseguer J., et al.: Maude manual (version 3.1), 2020. SRI International University of Illinois at Urbana-Champaign. <http://maude.lcc.uma.es/maude31-manual-html/maude-manual.html>.
- [10] Clavel M., Durán F., Hendrix J., Lucas S., Meseguer J., Ölveczky P.: The Maude formal tool environment. In: T. Mossakowski, U. Montanari, M. Haveraaen (eds.), *Algebra and Coalgebra in Computer Science: Second International Conference, CALCO 2007, Bergen, Norway, August 20–24, 2007. Proceedings*, pp. 173–178, Springer, 2007. doi: 10.1007/978-3-540-73859-6\_12.
- [11] Djaoui C., Kerkouche E., Chaoui A., Khalfaoui K.: A graph transformation approach to generate analysable maude specifications from UML interaction overview diagrams. In: *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pp. 511–517, IEEE, 2018. doi: 10.1109/iri.2018.00081.
- [12] Dobing B., Parsons J.: Dimensions of UML Diagram Use: A Survey of Practitioners, *Journal of Database Management (JDM)*, vol. 19(1), pp. 1–18, 2008. doi: 10.4018/jdm.2008010101.
- [13] Durán F., Eker S., Escobar S., Martí-Oliet N., Meseguer J., Rubio R., Talcott C.: Programming and symbolic computation in Maude, *Journal of Logical and Algebraic Methods in Programming*, vol. 110, 100497, 2020. doi: 10.1016/j.jlamp.2019.100497.
- [14] Eclipse Foundation: Acceleo, homepage, [Online]. <https://www.eclipse.org/acceleo/>. Accessed November 2023.

- [15] Eclipse Foundation: EMF, homepage Eclipse Modelling Framework (EMF), [Online]. <https://www.eclipse.dev/modeling/emf/>. Accessed November 2023.
- [16] Eclipse Foundation: Sirius, homepage, [Online]. <https://www.eclipse.org/sirius/>. Accessed November 2023.
- [17] Eker S., Meseguer J., Sridharanarayanan A.: The Maude LTL model checker, *Electronic Notes in Theoretical Computer Science*, vol. 71, pp. 162–187, 2004. doi: 10.1016/s1571-0661(05)82534-4.
- [18] Frick G., Scherrer B., Müller-Glaser K.D.: Designing the software architecture of an embedded system with UML 2.0. In: *Software Architecture Description & UML Workshop*, 2004.
- [19] Hammal Y.: A Formal Semantics of UML StateCharts by Means of Timed Petri Nets. In: F. Wang (ed.), *Formal Techniques for Networked and Distributed Systems – FORTE 2005. 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2–5, 2005, Proceedings*, pp. 38–52, Springer, 2005. doi: 10.1007/11562436\_5.
- [20] Kerkouche E., Khalfaoui K., Chaoui A.: A rewriting logic-based semantics and analysis of UML activity diagrams: a graph transformation approach, *International Journal of Computer Aided Engineering and Technology*, vol. 12(2), pp. 237–262, 2020. doi: 10.1504/ijcaet.2020.10026291.
- [21] Kloul L., Küster-Filipe J.: From Interaction Overview Diagrams to PEPA Nets, *Online Proceedings of the 4th Workshop on Process Algebras and Timed Activities (PASTA'05)*, vol. 104, 2005.
- [22] de Lara J., Vangheluwe H., Alfonseca M.: Meta-modelling and graph grammars for multi-paradigm modelling in ATOM<sup>3</sup>, *Software & Systems Modeling*, vol. 3, pp. 194–209, 2004. doi: 10.1007/s10270-003-0047-5.
- [23] Lilius J., Paltor I.P.: vUML: a Tool for Verifying UML Models. In: *14th IEEE International Conference on Automated Software Engineering*, pp. 255–258, 1999. doi: 10.1109/ASE.1999.802301.
- [24] Louati A., Jerad C., Barkaoui K.: On CPN-based verification of hierarchical formalization of UML 2 Interaction Overview Diagrams. In: *2013 5th International Conference on Modeling, Simulation and Applied Optimization (ICMSAO)*, pp. 1–6, IEEE, 2013. doi: 10.1109/icmsao.2013.6552703.
- [25] McUmber W.E., Cheng B.H.: A general framework for formalizing UML with formal languages. In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pp. 433–442, IEEE, 2001.
- [26] Meseguer J.: Rewriting Logic and Maude: A Wide-Spectrum Semantic Framework for Object-Based Distributed Systems. In: S.F. Smith, C.L. Talcott (eds.), *IFIP TC6/WG6.1. Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000) September 6–8, 2000, Stanford, California, USA*, pp. 89–117, Springer, 2000. doi: 10.1007/978-0-387-35520-7\_5.

- [27] Meseguer J.: Specifying, Analyzing and Programming Communication Systems in Maude. In: G. Hommel (ed.), *Communication-Based Systems: Proceeding of the 3rd International Workshop held at the TU Berlin, Germany, 31 March–1 April 2000*, pp. 93–101, Springer, 2000. doi: 10.1007/978-94-015-9608-4\_7.
- [28] Meseguer J.: Twenty years of rewriting logic, *The Journal of Logic and Algebraic Programming*, vol. 81(7–8), pp. 721–781, 2012. doi: 10.1016/j.jlap.2012.06.003.
- [29] Mishra A.: Dynamic Slicing of UML Interaction Overview Diagram. In: *2019 IEEE 9th International Conference on Advanced Computing (IACC)*, pp. 125–132, IEEE, 2019. doi: 10.1109/iacc48062.2019.8971586.
- [30] Padua D. (ed.): *Encyclopedia of Parallel Computing*, Springer Science & Business Media, New York, NY, 2011. doi: 10.1007/978-0-387-09766-4.
- [31] Platt R., Thompson N.: The evolution of UML. In: *Encyclopedia of Information Science and Technology*, Third Edition, pp. 348–353, IGI Global, 2015.
- [32] Rumbaugh J., Jacobson I., Booch G.: *The Unified Modeling Language Reference Manual*, 2nd Ed., Addison Wesley Longman Ltd., 2004.
- [33] Steinberg D., Budinsky F., Paternostro M., Merks E.: *EMF: eclipse modeling framework*, Pearson Education, 2008.
- [34] Störrle H., Hausmann J.H.: Towards a formal semantics of UML 2.0 activities. In: *Software Engineering 2005*, pp. 117–128, Gesellschaft für Informatik eV, 2005.
- [35] Wazlawick R.S.: *Object-oriented analysis and design for information systems: modeling with UML, OCL, and IFML*, Elsevier, 2014.
- [36] Whittle J.: Extending interaction overview diagrams with activity diagram constructs, *Software & Systems Modeling*, vol. 9, pp. 203–224, 2010.

## Affiliations

### Chafika Djaoui

Mohamed Seddik Ben Yahia University, Department of Computer Science, Jijel, Algeria;  
MISC Laboratory, Department of Computer Science and Its Applications, Constantine,  
Algeria, c.djaoui@univ-jijel.dz

### Allaoua Chaoui

University Constantine 2-Abdelhamid Mehri, MISC Laboratory, Department of Computer  
Science and Its Applications, Faculty of NTIC, Constantine, Algeria,  
allaoua.chaoui@univ-constantine2.dz

**Received:** 17.12.2023

**Revised:** 6.05.2024

**Accepted:** 22.05.2024