

Formal Modeling and Verification of UML Activity Diagrams (UAD) with FoCaLiZe

Messaoud Abbas^{a,*}, Renaud Rioboo^b, Choukri-Bey Ben-Yelles^c, Colin F. Snook^d

^aLABTHOP, College of Exact Sciences, El Oued University, 39000 Algeria

^bENSIIE, SAMOVAR, Square de la Résistance, F-91025 Evry, France

^cUniv. Grenoble Alpes, LCIS, Rue Barthélémy de Laffemas F-26901 Valence, France

^dECS, University of Southampton, Southampton, UK

Abstract

The UML Activity Diagram (UAD) is mostly used for modeling behavioral aspects of objects and systems. OCL (Object Constraint Language) is used together with UAD to specify guard conditions and action constraints. Due to the ambiguous semantics of UAD, it is relevant to formalize such diagrams using formal semantics and formal methods. In this paper, we opt for a formal transformation of UML activity diagrams denoted by functional semantics into FoCaLiZe, a proof based formal language. The ultimate goal is to detect eventual inconsistencies of UML activity diagrams and to prove their properties using Zenon, the automatic theorem prover of FoCaLiZe. In addition to the proposed formal basis for UAD. The presented approach directly supports action constraints, activity partitions and the communication between structural and dynamic aspects of UML models.

Keywords: UML Activity Diagram, UML Semantics, Software Engineering, Model Properties, Model Verification, Formal Methods.

1 Introduction

UML activity diagrams (UAD) [1] are graphical notations describing the behavior of UML class instances during their lifetime, without considering triggering events. They use the Object Constraint Language (OCL) [2] to specify action constraints and transition guards. UAD are frequently used in applications of workflow modeling [3], such as software application modeling [4], web services composition modeling [5] and also in business process modeling [6]. The wide use of UAD is enhanced with MDE (Model Driven Engineering) tools for verification and code generation.

The semantics of UAD models (especially within the scope of critical systems) lend themselves to combination with formal methods in order to express and check software properties using the verification techniques provided by formal methods.

Many studies have focused on the formalization and the verification of UAD. They use formal methods, such as B method [7], Alloy [8], Petri Nets [9], Maude [10] and model checkers

*Corresponding author

Email address: `messaoud-abbas@univ-eloued.dz` (Messaoud Abbas)

Preprint submitted to Elsevier

September 19, 2020

[11]. However, the large gap between UML (object oriented modeling) and formal methods (mathematical and logical specifications) can lead to a cumbersome model to model transformation that loses essential semantics of the original UML/OCL models.

We have already formalized most UML class diagram features [12, 13, 14], OCL constraints [15] and UML state machines [16] by transformation into FoCaLiZe [17]. The latter is a formal programming environment with a purely functional language that shares with UML and OCL most of their architecture, design and specification features [18, 13].

The transformation combines both the structural and behavioral aspects of UML models within a single FoCaLiZe model. Behavioral diagrams such as state machines communicate and refer to the structural elements of models, such as attributes, methods and OCL constraints. Thus, it is possible to express behavioral actions and transitions using references to attributes, class operation calls and class constraints. Note that some proofs of derived constraints can be achieved without the transformation of behavioral diagrams. Therefore, we transform the class diagram and its OCL constraint into FoCaLiZe before transformation of behavioral diagrams. The consistency of the whole model is ensured by the proof of all generated constraints using FoCaLiZe proof techniques.

In this paper, we contribute functional semantics of UAD in FoCaLiZe. This semantics is an extended description and specification of our work presented in [19].

Firstly, we propose a functional semantics for UAD that provides clear and effective solutions for choice control nodes (decision and merge nodes) and parallelism control nodes (fork and join nodes). Then we implement the proposed semantics using parallelism and choice statements in FoCaLiZe. In the final steps, we use FoCaLiZe proof techniques to check UAD properties. The proposed transformation considers OCL pre/post-conditions of classes operations and action constraints. It also supports activity partitions (activities that invoke several actors) and the communication between UML activity diagrams and their corresponding classes. Furthermore, the transformation of UAD into FoCaLiZe is realized in such a way that it could be naturally integrated with the proposals of the aforementioned UML class diagram transformation [18, 13].

The remainder of this document is organized as follows: sections 2 and 3 present basic concepts of FoCaLiZe and define the subset of UML activity diagram and OCL constraints supported by our transformation. In section 4, the semantics of UML activity diagrams is studied, followed by the specification of the transformation rules. Section 5 presents approaches that integrate UML classes, activity diagrams and the FoCaLiZe environment for error detection and proving of model properties. Section 6 describes the implementation of the transformation model and section 7 discusses some related works before concluding.

2 The FoCaLiZe Environment

FoCaLiZe [17] is an integrated development environment with formal features including a programming language, a constraint (property) specification language and theorem provers. A FoCaLiZe project is organized as a hierarchy of species that may have several roots. The upper levels are built along the specification stages, while the lower ones correspond to the implementation. Each node of the hierarchy corresponds to a refinement step toward a complete implementation using object oriented features such as multiple inheritance and parameterization.

The main brick in a FoCaLiZe project is the **species**, which groups together several methods: the carrier type of the species, functions to manipulate this carrier type and logical properties. The properties of a species express requirements that must be verified. Using EBNF (Extended Backus-Naur Form) notation, the general syntax of a species is:

Table 1: The Syntax of a Species

<i>spec</i>	::=	species <i>species_name</i> [(<i>param</i> [{ , <i>param</i> }*])] = [inherit <i>spec_def</i> [{ , <i>spec_def</i> *]]; { <i>methods</i> ;}* end ;;
<i>param</i>	::=	<i>ident in type</i> <i>ident is spec_def</i>
<i>spec_def</i>	::=	<i>species_name</i> <i>species_name</i> (<i>param</i> [{ , <i>param</i> }*])
<i>methods</i>	::=	<i>rep</i> <i>signature</i> <i>let</i> <i>property</i> <i>theorem</i>
<i>rep</i>	::=	representation = <i>type</i> ;
<i>signature</i>	::=	signature <i>function_name</i> : <i>function_type</i> ;
<i>let</i>	::=	let [rec] <i>function_name</i> = <i>function_body</i> ;
<i>property</i>	::=	property <i>property_name</i> : <i>property_specification</i> ;
<i>theorem</i>	::=	theorem <i>property_name</i> : <i>property_specification</i> proof = <i>theorem_proof</i> ;

A species is defined by a collection of methods as follows:

- The *representation* describes the data structure of the species entities.
- A *signature* specifies a function without giving its computational body, only the functional type is provided at this stage. A signature is intended to be defined (will get its computational body) later in the subspecies (through inheritances).
- A *let* defines a function together with its computational body.
- A *property* is a statement expressed by a first-order formula specifying requirements to be satisfied in the context of the species. A property is intended to be defined (will get its proof) later in the subspecies (through inheritances).
- A *theorem* is a property provided together with its formal proof.

The elements between parenthesis after the species name and after the inherit clause are needed for the parameterization and multiple inheritance mechanisms, which will be presented in the next sub-sections.

Code 1: The species Point

```

species Point =
signature getX : Self -> float;
signature getY : Self -> float;
signature equal: Self -> Self-> bool;
signature move : Self -> float -> float -> Self;
(* distance: calculates the distance between two given points *)
let distance (a:Self, b: Self):float = sqrt( ((getX(a) - getX(b))*(getX(a) - getX(b))) +
((getY(a) - getY(b))*(getY(a) - getY(b))) );
property equal_reflexive: all x: Self, equal (x, x) ;

property distanceSpecification: all p q:Self, equal(p, q) -> distance(p, q) = 0.0;
end::;

```

The species Point (see Code 1) models points of the plane. Each point is specified by its coordinates (the signatures `getX` and `getY`). The text enclosed between "*" and "*" represents comments. The key word `Self` refers to contextual instance (the one point that is the subject of this species), even though it is not yet instantiated. Because the representation of the species Point is still undefined, it is possible to inherit from this species to construct new species with different representations.

2.1 Inheritance

When creating a species by multiple inheritance, some signatures can be instantiated by functions and properties by theorems. It is also possible to associate a *definition* of function to a

signature, a proof to a property or to redefine a method even if it is already used by an existing method. All these features are due to the FoCaLiZe late-binding mechanism.

A species is said to be *complete* if all declarations have received definitions and all properties have received proofs. The representations of complete species are encapsulated through species interfaces. The interface of a complete species is the list of its function types and its properties. It corresponds to the end user point of view, who needs only to know which functions he can use, and which properties these functions satisfy, but doesn't care about the details of the implementation. A collection is created by the implementation of a complete species. A collection can hence be seen as an abstract data type, only usable through the methods of its interface.

The species `ColoredPoint` (see Code 2) aims to manipulate colored (graphical) points. Note that the functions `fst` and `snd` (predefined functions) return, respectively, the first and the second component of a pair.

Code 2: The species `ColoredPoint`

```
(* Definition of the type color *)
type color = | Red | Green | Blue ;;

species ColoredPoint = inherit Point;
representation = (float * float) * color;
let getColor(p:Self):color = snd(p);
let newColoredPoint(x:float, y:float, c:color): Self = ((x, y), c);
let getX(p) = fst(fst(p));
let getY(p) = snd(fst(p));
let move(p, dx, dy) = newColoredPoint(getX(p) + dx, getY(p) + dy, getColor(p));
let equal(p:Self, q:Self) = (getX(p) = getX(q)) && (getY(p) = getY(q)) && (getColor(p) = getColor(q));
let printPoint (p:Self):string = let printColor (c:color) = match c with
  | Red -> "Red"
  | Green -> "Green"
  | Blue -> "Blue"
  in (" X = " ^ string_of_float(getX(p)) ^
      " Y = " ^ string_of_float(getY(p)) ^
      " COLOR = " ^ printColor(getColor(p)));
proof of distanceSpecification = by definition of equal, distance;
proof of equal_reflexive = assumed;
end::;

collection ColoredPointCollection = implement ColoredPoint; end ;;
let p = ColoredPointCollection!
newColoredPoint(2.0, 5.0, Blue);;
basics#print_string (ColoredPointCollection!printPoint(p));;
```

At the top level (outside the species), we define a given type `color` that will be used by the species `ColoredPoint`. The latter is a complete species that inherits the species `Point` and provides definitions (computational bodies) for all its signatures and proofs for all its properties (including inherited signatures and properties). Some new methods are also added (`getColor`, `newColoredPoint` and `printPoint`) in order to get the color of a point, to allow the creation of new instances and to print the coordinates and the color of a given point.

The collection `ColoredPointCollection` implements the complete species `ColoredPoint`. Then, it is used to create a new entity, `p`, of the species `ColoredPoint` and print its coordinates.

The “!” notation (in Code 2) is equivalent to the usual dot notation of message sending in object-oriented programming.

2.2 Parameterization

Parameterization specifies a (supplier-client) dependency relationship between species. Via this mechanism, a species (the client) can use the methods of other species (the suppliers) in order to develop its own methods.

In mathematics, a circle is defined by its radius and its center (a point). The species `Circle` (see Code 3) is parameterized using the species `Point` in order to define the center of a circle.

Code 3: The species `Circle`

```

species Circle (P is Point) =
representation = P * float ;
let newCircle(centre:P, radius:float):Self = (centre , radius);
let getCenter(c:Self):P = fst(c);
let getRadius(c:Self):float = snd(c);
let belongs(p:P, c:Self):bool = (P!distance(p, getCenter(c))= getRadius(c));
theorem belongs_specification : all c:Self, all p:P,
  belongs(p, c) <-> (P!distance(p, getCenter(c))= getRadius(c))
proof =
<1>1 assume c : Self, p : P,
  prove belongs(p, c) <-> (P!distance(p, getCenter(c))= getRadius(c))
<2>1 hypothesis h1 : belongs(p, c),
  prove belongs(p, c) -> (P!distance(p, getCenter(c))= getRadius(c))
  by hypothesis h1 definition of belongs
<2>2 hypothesis h2 : (P!distance(p, getCenter(c))= getRadius(c)),
  prove (P!distance(p, getCenter(c)) = getRadius(c)) -> belongs(p, c)
  by hypothesis h2 definition of belongs property basics#beq_symm
<2>3 qed by step <2>1, <2>2
<1>2 conclude ;
end::;

```

Here, the species `Circle` can use all signatures, functions and properties of `Point`, even if they are not completely defined yet.

2.3 Proofs and Compilation

FoCaLiZe provides several means to write the proofs of properties. We can directly write Coq proofs or use the key word **assumed** to avoid providing proofs. However, the usual way to write proofs is to use the FoCaLiZe proof language (FPL) (see the proof of the theorem `belongs_specification` of the species `Circle`, Code 3). Using FPL, the developer organizes the proof in steps. Each step provides proof hints that will be exploited by Zenon (the automatic theorem prover of FoCaLiZe) [20].

Finally, the compilation of FoCaLiZe sources produces OCaml and Coq [21] code. The OCaml code provides the executable program. The Coq code is automatically generated by Zenon, when it succeeds finding proofs. The Coq code is then checked by the Coq theorem prover.

3 Abstract Syntax for UAD

UML is a general-purpose modeling language that helps developers to design, visualize and document the artifacts of software engineering. A UML model is a set of diagrams describing the static and the behavioral aspects of software systems. Thanks to the declarative language OCL (Object Constraint Language) one can describe constraints (properties) of UML models. An OCL constraint is a precise statement which may be attached to any UML element.

An activity diagram (of a UML class) describes the behavior of the class objects. It specifies the sequence of actions (workflow) of an object during its lifetime. For clarity sake, we only focus on simple actions (class operation calls) in this paper.

In order to provide a formal framework for the transformation of UAD to FoCaLiZe specifications, we propose an abstract syntax for the subset of UAD constructs that we consider, using mostly **UML metamodel syntax** [1]. The UAD subset that we support is sufficient to express any behavior of a UML class diagram. However the additional UAD features that are not yet supported allow such models to be constructed with more structure. Note that, the UAD/OCL

Table 2: Abstract Syntax of UML Activity Diagrams

<i>ActivityDiagram</i>	::=	<i>ActivityDiagramIdent</i> <i>declaration</i> *
<i>declaration</i>	::=	<i>node</i> <i>transition</i>
<i>node</i>	::=	<i>sourceNode</i> <i>targetNode</i>
<i>sourceNode</i>	::=	InitialNode <i>actionNode</i> <i>controlNode</i>
<i>targetNode</i>	::=	FinalNode <i>actionNode</i> <i>controlNode</i>
<i>actionNode</i>	::=	<i>nodeIdent</i> [« localprecondition » <i>Pre-Condition</i>] <i>operation_call</i> [« localpostcondition » <i>Post-Condition</i>]
<i>controlNode</i>	::=	<i>nodeIdent</i> { DecisionNode ForkNode JoinNode MergeNode } ¹
<i>operation_call</i>	::=	<i>self.operationIdent</i> ([<i>actualParameter</i> { , <i>actualParameter</i> }*])
<i>guard</i>	::=	<i>OclExpression</i>
<i>Pre-Condition</i>	::=	<i>OclExpression</i>
<i>Post-Condition</i>	::=	<i>OclExpression</i>
<i>transition</i>	::=	transition <i>transitionIdent</i> (<i>sourceNodeIdent</i> [[<i>guard</i>]] <i>targetNodeIdent</i>)

elements are clearly defined for the purposes of translation but not in a mathematical form that enables us to reason about the model. The proposed syntax is written in EBNF notation (see Table 2) in order to improve the readability of our transformation rules.

In the syntax presented in Table 2, an activity diagram consists of a set of nodes and a set of transitions. Each transition (flow) is a directed edge interconnecting two nodes. It specifies that the system moves from one action to another. We distinguish between two types of nodes: control nodes and action nodes.

Control nodes (see Fig. 1) are used to specify choice (**DecisionNode** and **MergeNode**), parallelism (**ForkNode** and **JoinNode**), initial nodes (**InitialNode**) or final nodes (**FinalNode**).

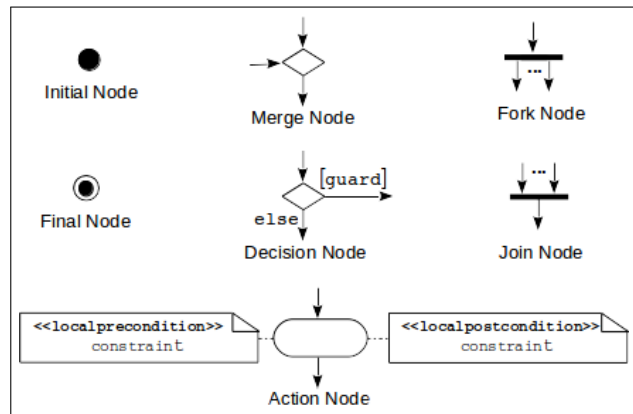


Figure 1: UML activity diagram notations

The decision nodes choose one of the outgoing transitions. The merge nodes merge several incoming transitions so that the first incoming transition will become the first outgoing one. The Join nodes synchronize several parallel incoming transitions, while the fork nodes split a transition into several parallel outgoing flows.

Action nodes are calls to class operations (*operation_call*). The initial node starts the global flow.

Action constraints are presented as notes attached to action nodes, specified with the stereotypes «**localprecondition**» and «**localpostcondition**». The local pre-condition describes a constraint which is assumed to be true before the action is executed. The local post-condition

describes a constraint which has to be satisfied after the action is executed. We use OCL syntax to specify local pre and post-conditions on actions.

A transition guard (*[guard]*) is an optional constraint that controls the firing of the transition. If the guard is true, the transition may be enabled, otherwise it is disabled.

An expression of the OCL language [2] uses types and operations on types. We distinguish between primitive (Integer, Boolean, Real and String), enumeration type, object type (classes of UML model) and collection types. For a given OCL type, T, the OCL type Collection(T) represents a collection family of elements of type T.

In this work, we have considered the two main OCL constraints: class invariants and pre and post-conditions on classes operations. An invariant is an OCL expression attached to one class (c_n) and must be true for all instances of that class at any time. Its general form is:

context c_n inv : \mathbb{E}_{inv}

where \mathbb{E}_{inv} is the OCL expression describing the invariant.

The pre-condition of an operation OP_n of the class c_n describes a constraint which is assumed to be true before the operation is executed. The post-condition of OP_n describes a constraint which has to be satisfied after the operation is executed:

context c_n :: $OP_n(p_1 : typeExp_1 \dots p_m : typeExp_m)$ **pre** : \mathbb{E}_{pre} **post** : \mathbb{E}_{post}

where $p_1 \dots p_m$ are the operation parameters, $typeExp_1 \dots typeExp_m$ their corresponding types and \mathbb{E}_{pre} and \mathbb{E}_{post} are the OCL expressions describing the pre/post-conditions.

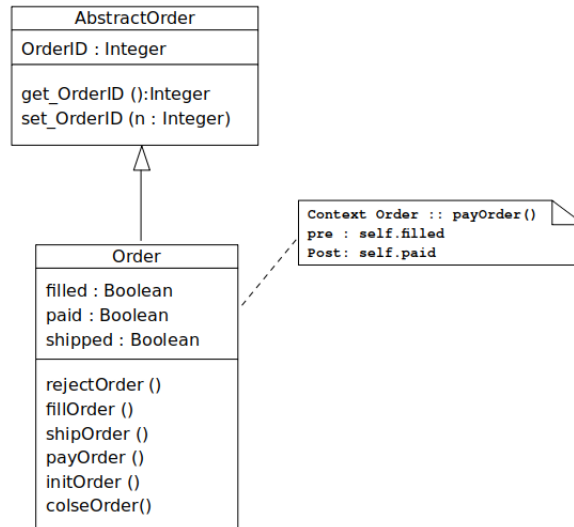


Figure 2: The class Order

Activity Diagram Example: Orders Processing.

We present here the activity diagram of the class Order (see Fig. 2 and Fig. 3).

Through inheritance, the class Order concretizes and refines the abstract class AbstractOrder, which cannot be instantiated. It acquires all attributes and methods of the class AbstractOrder and defines its own attributes and methods.

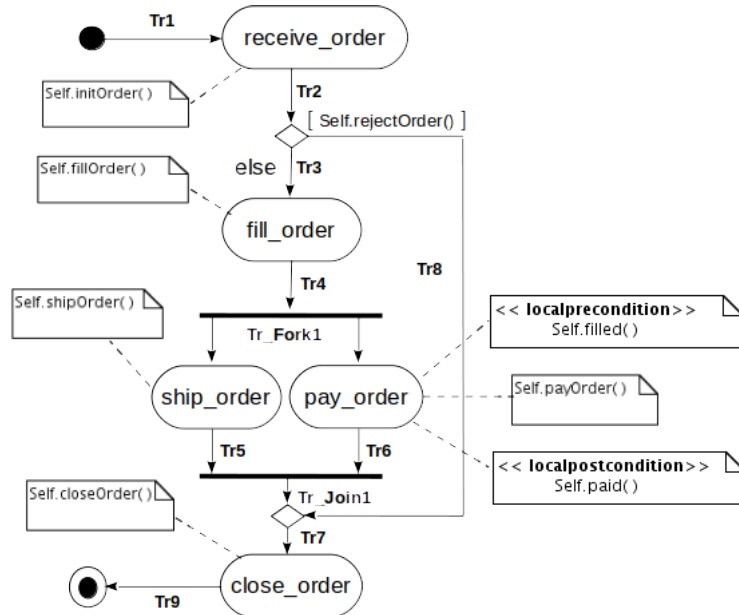


Figure 3: UML Activity Diagram: orders processing

Firstly, an order (a new instance o of the class `Order` is created) is received by the action `receive_order` (the operation `initOrder()` of the class `Order` is invoked). Then, if the condition guard (`Self.rejectOrder()`) of the decision node is not true (the order is not rejected), the flow goes to the next step: the action `fill_order` (`Self.fillOrder()`). After that, the fork node splits the path of the control flow into two parallel tasks. On the left path, the action `ship_order` (call of the operation `shipOrder()`) is executed. On the right path, to bill the order and process its payment, the action `pay_order` (`Self.payOrder()`) is handled. When the two paths are accomplished, the join node may take place and the action `close_order` (`Self.closeOrder()`) is achieved. Returning back to the above decision node, if the order is rejected, the flow is passed directly to the action `close_order`.

4 From UAD to FoCaLiZe

A UML activity diagram describes the behavior of a UML class and uses OCL expressions. Therefore, we will first present an overview on the transformation of UML classes and OCL expressions, then we describe the semantics and transformation rules for activity diagrams.

During the transformation from UML/OCL to FoCaLiZe, we will use the following notations:

- The term “class c_n ” refers to the UML class named c_n and the term “species c_n ” refers to the FoCaLiZe species named c_n .
- For a UML/OCL element E , $[[E]]$ denotes its transformation into FoCaLiZe. We will also preserve the same UML/OCL element identifiers in the transformations, taking into account upper and lower cases to respect FoCaLiZe syntax if necessary.

4.1 Transformation of UML Classes and OCL Constraints

The similarities between FoCaLiZe species and UML classes [22] led us to transform a UML class into a FoCaLiZe species.

Attributes specify the state of class objects. Therefore, each attribute gives rise to a signature modeling its getter function in the corresponding species:

UML:

attrName : *typeExp* [*mult*];

FoCaLiZe:

signature *get_attrName* : Self -> [[*typeExp* [*mult*]]];

Operations represent services invoked by any object of the class in order to affect object behaviors. In the context of object oriented programming languages, when an instance *o* of the class *c_n* invokes an operation named *N* of the class, the memory state of the instance *o* is affected and moves to a new memory state *o'*. In functional languages (without memory state) such as FoCaLiZe, the two memory states of an object represent two different entities. Taking into account this difference between the two formalisms, we convert a class operation into a species signature (function interface) that starts with the type Self (the entity that invokes the function), followed by the function parameter types, and ends with the type Self (the new created entity). So, the general transformation of operations is:

UML:

$$N \left(\begin{array}{l} dir_1 p_1 : type_1 [mult_1] \\ \dots \\ dir_k p_k : type_k [mult_k] \end{array} \right) : Type[mult]$$

FoCaLiZe:

signature *N*: [Self]->[[*Type*₁[*mult*₁]]->...-> [[*Type*_{*k*}[*mult*_{*k*}]]->[Self];

OCL constraints are mapped into species properties.

To facilitate the transformation of OCL expressions we built a FoCaLiZe library that formalizes OCL expressions. In this library, classes of the UML model correspond to FoCaLiZe species and OCL primitive types (Integer, Real, String and Boolean) correspond to FoCaLiZe primitive types (int, float, string and bool). OCL collection types are handled using the general species OCL_Collection of the library, in which we specify functions for OCL operations on collections (forall, isEmpty, size...). The OCL constraints (invariants, pre-conditions and post-conditions) specified in the context of a UML class are then mapped into FoCaLiZe properties of the corresponding species.

Most of the OCL expressions on types Integer, String and Real are directly converted into their corresponding FoCaLiZe expressions using almost the same operations on FoCaLiZe types: int, string and float.

Most OCL formulas (of type Boolean, see Table 3) have a straightforward counterpart as FoCaLiZe boolean expressions. In Table 3, ϕ and ψ are two OCL formulas, α and β are two numeric (integer/real) expressions, *o* is an object of the model and *coll*, *coll*₁ and *coll*₂ are OCL collections.

Note that, the OCL operations forall and exists, when applied to the OCL collection returned by the allInstances operation, are respectively mapped to the FoCaLiZe universal (all) and existential (ex) quantifiers. Otherwise, they must be turned into special defined FoCaLiZe operations that iterate on all instances of a collection.

Table 3: Mapping of OCL formulas

OCL	FoCaLiZe
true	true
false	false
not (ϕ)	$\sim\llbracket\phi\rrbracket$
ϕ and ψ	$\llbracket\phi\rrbracket \wedge \llbracket\psi\rrbracket / \llbracket\phi\rrbracket \ \&\& \ \llbracket\psi\rrbracket$
ϕ or ψ	$\llbracket\phi\rrbracket \ \vee \ \llbracket\psi\rrbracket / \llbracket\phi\rrbracket \ \vee \vee \ \llbracket\psi\rrbracket$
ϕ xor ψ	$\llbracket\phi\rrbracket \ \vee \ \llbracket\psi\rrbracket \ \wedge \ \sim(\llbracket\phi\rrbracket \ \wedge \ \llbracket\psi\rrbracket)$
ϕ implies ψ	$\llbracket\phi\rrbracket \ \rightarrow \ \llbracket\psi\rrbracket$
if ϕ then ψ else ϕ	if $\llbracket\phi\rrbracket$ then $\llbracket\psi\rrbracket$ else $\llbracket\phi\rrbracket$
let $x : type = Exp$ in ϕ	let $x = \llbracketExp\rrbracket$ in $\llbracket\phi\rrbracket$
$\alpha = \beta$	$\llbracket\alpha\rrbracket = \llbracket\beta\rrbracket$
$\alpha < > \beta$	$\sim\sim(\llbracket\alpha\rrbracket = \llbracket\beta\rrbracket)$
$\alpha > \beta$	$\llbracket\alpha\rrbracket > \llbracket\beta\rrbracket$
$\alpha < \beta$	$\llbracket\alpha\rrbracket < \llbracket\beta\rrbracket$
$\alpha >= \beta$	$\llbracket\alpha\rrbracket >= \llbracket\beta\rrbracket$
$\alpha <= \beta$	$\llbracket\alpha\rrbracket <= \llbracket\beta\rrbracket$
allInstances \rightarrow forAll ($x \mid \phi$)	all $x : \text{Self}, \llbracket\phi\rrbracket$
allInstances \rightarrow exists ($x \mid \phi$)	ex $x : \text{Self}, \llbracket\phi\rrbracket$
$coll \rightarrow$ forAll ($x \mid \phi$)	forAll ($\llbracketcoll\rrbracket, \llbracket\phi\rrbracket$)
$coll \rightarrow$ exists ($x \mid \phi$)	exists ($\llbracketcoll\rrbracket, \llbracket\phi\rrbracket$)
$coll \rightarrow$ isEmpty ()	isEmpty ($\llbracketcoll\rrbracket$)
$coll \rightarrow$ notEmpty ()	notEmpty ($\llbracketcoll\rrbracket$)
$coll \rightarrow$ includes (o)	includes ($\llbracketo\rrbracket, \llbracketcoll\rrbracket$)
$coll \rightarrow$ excludes (o)	excludes ($\llbracketo\rrbracket, \llbracketcoll\rrbracket$)
$coll_1 \rightarrow$ includesAll ($coll_2$)	includesAll ($\llbracketcoll_1\rrbracket, \llbracketcoll_2\rrbracket$)
$coll_1 \rightarrow$ excludesAll ($coll_2$)	excludesAll ($\llbracketcoll_1\rrbracket, \llbracketcoll_2\rrbracket$)

An OCL invariant \mathbb{E}_{inv} of the class c_n is converted into a FoCaLiZe property of the corresponding species:

OCL:

```
context c_n inv :  $\mathbb{E}_{inv}$ 
```

FoCaLiZe:

```
property inv_ident : all  $e : \text{Self}, \llbracket\mathbb{E}_{inv}\rrbracket ;$ 
```

OCL pre and post-conditions \mathbb{E}_{pre} and \mathbb{E}_{post} of an operation op_n of the class c_n specify that the post-condition is satisfied after the operation execution, when the pre-condition is satisfied before the operation execution. Therefore, we convert pre and post-conditions together into a FoCaLiZe implication of the corresponding species:

OCL:

```
context  $c\_n :: op\_n(p_{1\_n} : typeExp_1 \dots p_{k\_n} : typeExp_k)$ 
```

```
  pre :  $\mathbb{E}_{pre}$ 
```

```
  post :  $\mathbb{E}_{post}$ 
```

FoCaLiZe:

```
property pre_post_ident :
```

```
  all  $e : \text{Self},$ 
```

```
  all  $p_{1\_n} : \llbracket typeExp_1 \rrbracket, \dots, \text{all } p_{k\_n} : \llbracket typeExp_k \rrbracket,$ 
```

```
   $\llbracket\mathbb{E}_{pre}(e)\rrbracket \rightarrow \llbracket\mathbb{E}_{post}(e')\rrbracket ;$ 
```

where $\llbracket\mathbb{E}_{pre}(e)\rrbracket$ is the transformation of the pre-condition applied to the entity e (before the operation is executed) and $\llbracket\mathbb{E}_{post}(e')\rrbracket$ is the transformation of the post-condition applied to the entity e' resulting from the operation op_n (after the operation execution). The *inv_ident* and *pre_post_ident* are identifiers that are assigned to invariants and pre/post-conditions.

To transform (multiple) inheritance, templates, template bindings and dependency features, we use similar mechanisms in FoCaLiZe. Although FoCaLiZe is a functional language, it supports multiple inheritance, parameterized species (like templates in UML) and the substitution of formal parameters of species, which is similar to template binding in UML.

This paper focuses more on the formalization of UAD, so for more details about the formal transformation of classes attributes, methods and OCL constraints, the reader may refer to [12, 13, 15, 14].

Code 4 illustrates the transformation of the abstract classes `AbstractOrder` and its subclass `Order` (see Fig. 2) and its OCL constraints. The transformation generates the species `AbstractOrder` and `Order` using the inheritance mechanism in FoCaLiZe, which accurately reflects inheritances in UML/OCL.

Code 4: The species `Order`, derived from the class `Order`

```

species AbstractOrder =
signature get_OrderId : Self -> int ;
signature set_OrderId : int -> Self ;
end;;

species Order = inherit AbstractOrder;
(* Transformation of attributes *)
signature get_filled : Self -> bool;
signature get_shipped : Self -> bool;
signature get_paid : Self -> bool;

(* Transformation of operations *)
signature newOrder : bool -> bool -> bool -> Self;    (*The species constructor*)
signature rejectOrder : Self -> bool ;
signature fillOrder : Self -> Self;
signature shipOrder : Self -> Self;
signature payOrder : Self -> Self;
signature initOrder : Self -> Self ;
signature closeOrder : Self -> Self ;

(* Transformation of payOrder pre/post-constraints *)
property payOrder_pre_post: all e:Self, (get_filled(e)) -> (get_paid(payOrder(e)));
end;;

```

Note that the species constructor is always automatically generated, even if it is not explicitly declared in the original class.

4.2 Functional semantics for UAD and its transformation to FoCaLiZe

Let us start with the description of a functional semantics for a simple control flow (without emphasis on the sequence and conditions of the flow). We will then present the implementation of this semantics in FoCaLiZe. After this, we will describe the transformation of control nodes (decision, merge, fork and join nodes) using corresponding functional statements in FoCaLiZe.

Based on the previously described syntax for UAD (see Table 2), an activity diagram $\mathbb{A}\mathbb{D}$ of a class c_n consists of a set of nodes ($\mathbb{N}\mathbb{D}$) and a set of transitions ($\mathbb{T}\mathbb{R}$):

$$\begin{aligned} \mathbb{N}\mathbb{D} &= \{\text{InitialNode}, nd_1, \dots, nd_k, \text{FinalNode}\} \\ \mathbb{T}\mathbb{R} &= \{tr_1, \dots, tr_m\}. \end{aligned}$$

Each action node nd has form:

$$\begin{aligned} nd = \text{nodeIdent} [\llbracket \text{localprecondition} \rrbracket \text{Pre-Condition}] \\ \text{self.action} (\text{param}_1 \dots \text{param}_n) \\ [\llbracket \text{localpostcondition} \rrbracket \text{Post-Condition}] \end{aligned}$$

where $self.action$ ($param_1 \dots param_n$) is an operation call of the class c_n .

Thus, the set of node identifiers is:

$$\mathbb{NI} = \{ \text{InitialNode}, \text{FinalNode}, \text{nodeIdent}_1, \dots, \text{nodeIdent}_k \}.$$

Each transition tr has form:

$$tr = \mathbf{transition} \ trIdent \ (sourceNode \ [guard] \ targetNode)$$

where the pair $(sourceNode, targetNode) \in \mathbb{NI}^2$ represents the source and target node identifiers (in addition to `InitialNode` and `FinalNode`, we only have action nodes at this step) of the transition tr .

The optional transition guard ($[guard]$) guards the transition. It is a logical formula specified by an OCL expression.

The set of transition identifiers is $\mathbb{TI} = \{ trIdent_1, \dots, trIdent_m \}$.

Let \mathbb{I} be the list of instances (snapshots) of the class c_n . For an instance o , $state(o)$ denotes its current action node in the activity diagram of the class c_n .

$$\begin{aligned} state : \mathbb{I} &\rightarrow \mathbb{NI} \\ (o) &\mapsto state(o) \end{aligned} \tag{1}$$

An activity diagram of the class c_n can be thought of as a **function** that takes a transition tr and an instance $o \in \mathbb{I}$ such that $state(o) = sourceNode$ (the source node identifier of the transition tr) as parameters and returns the instance o' such that $state(o') = targetNode$ (the target node identifier of the transition tr). The instance $o' = o.action(param_1 \dots param_n)$ is the result of the execution of the action (operation call) in the source action node ($sourceNode$):

$$\begin{aligned} AD : \mathbb{I} \times \mathbb{TI} &\rightarrow \mathbb{I} \\ (o, tr_i) &\mapsto o' \end{aligned} \tag{2}$$

In the context of object oriented programming languages o and o' are the same object (instance), but with different memory states. In functional languages (without memory states) such as FoCaLiZe, o and o' are two different entities, which makes the above, functional, interpretation of activity diagrams perfectly suited for transformation to FoCaLiZe.

Using the functional semantics presented in (2), the activity diagram of one class directly communicates with the structural components of the same class and other classes of the model that connected with relationships. In such a way, all behavior actions are directly expressed using class operation calls.

The implementation of the UAD of the class c_n into FoCaLiZe translates the function AD (modeling the activity diagram, see formula (2)). It is a function aD (function names must start with a lowercase letter, in FoCaLiZe syntax) of the species c_n derived from the class c_n .

Before describing the function aD , let us first highlight the following correspondences between UML and FoCaLiZe:

- The class c_n is transformed into a species having the same name (c_n).
- The set of instances of the class c_n (\mathbb{I}) corresponds to the set of entities of the species c_n (\mathbb{E}).
- The set of transition identifiers \mathbb{TI} is modeled by a new FoCaLiZe enumeration (`transitions`).
- The set of node identifiers \mathbb{NI} is modeled by a new FoCaLiZe enumeration (`nodes`).

- The function *state* (see formula (1)) is transformed into the species signature state:
signature state: Self -> nodes
- The instances (memory states) *o* and *o'* correspond to the species entities *e* and *e'* ($e' = \llbracket o.action(param_1 \dots param_n) \rrbracket$, the operation call holds in the node *sourceNode*).

Table 4: General transformation of UAD into FoCaLiZe

<p>UML:</p> <pre>public class c_n = \mathbb{A} ; \odot end</pre> $\left(\begin{array}{l} nd_1[\langle\langle\text{localprecondition}\rangle\rangle PreCondition_1] self.action_1(prm_{11} \dots prm_{1n}) \\ [\langle\langle\text{localpostcondition}\rangle\rangle PostCondition_1] \\ \dots \\ nd_k[\langle\langle\text{localprecondition}\rangle\rangle PreCondition_k] self.action_k(prm_{k1} \dots prm_{kn}) \\ [\langle\langle\text{localpostcondition}\rangle\rangle PostCondition_k] \end{array} \right)$ $\left(\begin{array}{l} \text{transition } tr_1(sourceNode_1 \llbracket guard_1 \rrbracket targetNode_1) \\ \dots \\ \text{transition } tr_m(sourceNode_m \llbracket guard_m \rrbracket targetNode_m) \end{array} \right)$
<p>FoCaLiZe</p> <pre>type nodes = $\llbracket nd_1 \rrbracket \dots \llbracket nd_k \rrbracket$;; type transitions = $\llbracket tr_1 \rrbracket \dots \llbracket tr_m \rrbracket$;; species c_n = ... signature state: Self -> nodes ; signature $\llbracket action_1 \rrbracket$: Self -> $\llbracket prmType_{11} \rrbracket \dots -> \llbracket prmType_{1n} \rrbracket -> Self$; ... signature $\llbracket action_k \rrbracket$: Self -> $\llbracket prmType_{k1} \rrbracket \dots -> \llbracket prmType_{kn} \rrbracket -> Self$; let aD(t:transitions, e:Self): Self = match t with $\llbracket tr_1 \rrbracket$ -> if ($\llbracket guard_1 \rrbracket \wedge (state(e) = \llbracket sourceNode_1 \rrbracket)$) then e else focalize_error ("ERROR"); $\llbracket tr_i \rrbracket$ -> if ($\llbracket guard_i \rrbracket \wedge (state(e) = \llbracket sourceNode_i \rrbracket)$) then $\llbracket action_{sn_i}(e, prm_{sn_i1}, \dots, prm_{sn_in}) \rrbracket$ else focalize_error ("ERROR"); $\llbracket tr_m \rrbracket$ -> if ($\llbracket guard_m \rrbracket \wedge (state(e) = \llbracket sourceNode_m \rrbracket)$) then $\llbracket action_{sn_m}(e, prm_{sn_m1}, \dots, prm_{sn_mn}) \rrbracket$ else focalize_error ("ERROR"); _ -> focalize_error("ERROR") ; end;;</pre>

So, on the FoCaLiZe side, for a given entity *e* of the species *c_n* (derived from the class *c_n*) and for a given transition identifier $\llbracket trId \rrbracket$, the function aD returns a new entity *e'*:

$$aD: \mathbb{E} \times \text{transitions} \rightarrow \mathbb{E} \quad (3)$$

$$(e, \llbracket trId \rrbracket) \mapsto e'$$

In order to group the transformation of all transitions of one activity diagram within a single function (see Table 4), we use the **pattern matching** in FoCaLiZe. So, each time we call the function `aD` only one pattern will be invoked. This is, of course, according to the state of the entity passed as parameter. If no pattern matches the state of the entity, the function will return an error message that expresses a **system deadlock**.

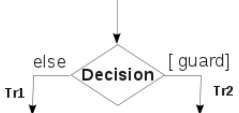
When we call the transformation function (`aD`) using the parameters tr_1 (the first transition in the UAD) and an entity e , such that $state(e) = \text{InitialNode}$ (It is always the source state of the transition tr_1), the transition is fired and returns as result the same entity e with a new state (the next node in the UAD).

4.2.1 Transformation of control nodes

Using the transformation presented in Table 4, the `DecisionNode` and the `MergeNode` have a straightforward counterpart as FoCaLiZe expressions.

However, new functions need to be defined for the `ForkNode` and the `JoinNode`. for each category of control nodes (one function for fork nodes and one function for join nodes).

Table 5: General Transformation of Decision Nodes

<p>UML:</p>  <pre> (Decision DecisionNode ... (transition tr1(Decision[[not(guard)]])...) transition tr2(Decision[[guard]]...)) </pre> <p>FoCaLiZe</p> <pre> type nodes = [[Decision]] ... ;; type transitions = [[tr1]] [[tr2]] ... ;; species c_n = ... signature state: Self -> nodes ; let aD(t:transitions, e:Self): Self = match t with [[tr1]] -> if ([[not(guard)]] ^ (state(e)= [[Decision]])) then e else focalize_error ("ERROR"); [[tr2]] -> if ([[guard]] ^ (state(e)= [[Decision]])) then e else focalize_error ("ERROR"); ... end;; </pre>

DecisionNode:

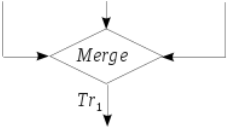
A decision node (see Table 5) is followed by two transitions. If the first transition is guarded by the guard $[guard]$, the second transition is guarded by its negation ($\mathbf{not}([guard])$). In this paper, we only support decision nodes followed by two transitions.

A decision node followed by more than two transitions should be transformed into several decision nodes each followed by two transitions before transformation. Therefore, the two transitions are naturally handled using the above function \mathbf{aD} .

MergeNode:

The formalization of merge nodes consists in directing (guiding) the processing of all incoming transitions (of the merge node) to the pattern modeling the outgoing transition in the function \mathbf{aD} (see Table 6).

Table 6: General Transformation of Merge Nodes

<p>UML:</p>  <p>$($ Merge MergeNode $)$ $($... $)$ $($ transition tr_1(Merge $[[true]]$)... $)$</p> <p>FoCaLiZe type nodes = $[[Merge]]$... type transitions = $[[tr_1]]$... ;;</p> <p>species c_n = ... signature state: Self -> nodes ;</p> <p>let $\mathbf{aD}(t:\text{transitions}, e:\text{Self}): \text{Self} =$ match t with $[[tr_1]]$ -> if ((state(e) = $[[Merge]]$)) then e else focalize_error ("ERROR"); ... end;;</p>
--

ForkNode:

For the transformation of fork nodes we need to define a particular function (ADFN) that should generate as many copies of the result object as the outgoing transitions (paths) from the fork node:

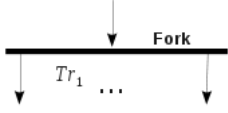
$$\begin{aligned} ADFN : \mathbb{I} \times \mathbb{T}\mathbb{I} &\rightarrow \mathbb{I} \times \dots \times \mathbb{I} \\ (o, tr_i) &\mapsto (o, \dots, o) \end{aligned} \tag{4}$$

For this purpose, we use the FoCaLiZe type `list(t)` that groups several elements of the same type `t`. So, on the FoCaLiZe side, for a given entity e of the species c_n (derived from the class c_n) and for a given transition identifier $\llbracket trId \rrbracket$, the function `aDFN` (that transforms the function $ADFN$) returns a list of entities `list(E)`:

$$\begin{aligned} \text{aDFN} : \mathbb{E} \times \text{transitions} &\rightarrow \text{list}(\mathbb{E}) \\ (e, \llbracket trId \rrbracket) &\mapsto [e; \dots; e] \end{aligned} \quad (5)$$

Table 7 summarizes the general transformation of fork nodes.

Table 7: General Transformation of Fork Nodes

<p>UML:</p>  <p>$(\text{Fork } \text{ForkNode})$ \dots transition $tr_1(\text{Fork}[\llbracket true \rrbracket]) \dots$</p> <p>FoCaLiZe</p> <pre> type nodes = $\llbracket Fork \rrbracket$... ;; type transitions = $\llbracket tr_1 \rrbracket$... ;; species c_n = ... signature state: Self -> nodes ; let aDFN(t:transitions, e:Self): list(Self) = match t with $\llbracket tr_1 \rrbracket$ -> if (state(e) = $\llbracket Fork \rrbracket$) then [e; ...; e] else focalize_error ("ERROR"); ... _ -> focalize_error("ERROR") ; end;</pre>

JoinNode:

Similarly, we need to define a particular function ($ADJN$) for join nodes. The function $ADJN$ has a collection of objects of the class c_n and a transition as parameters and returns one object of the class:

$$\begin{aligned} ADJN : (\mathbb{I} \times \dots \times \mathbb{I}) \times \mathbb{T}\mathbb{I} &\rightarrow \mathbb{I} \\ ((o_1, \dots, o_n), tr) &\mapsto o \end{aligned} \quad (6)$$

To transform the function $ADJN$ to FoCaLiZe, we also use the type `list(t)`. So, for a given list of entities of the species c_n (derived from the class c_n) and for a given transition identifier

$\llbracket trId \rrbracket$, the function `aDJN` (that transforms the function `ADJN`) returns one entity of the species:

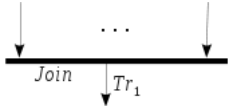
$$\begin{aligned} \text{aDFN} : \text{list}(\mathbb{E}) \times \text{transitions} &\rightarrow \mathbb{E} \\ ([e_1; \dots; e_n], \llbracket trId \rrbracket) &\mapsto e \end{aligned} \quad (7)$$

In Addition, we use the function (signature) `join()` specified in the species `c_n` as follows:
signature `join : list(Self) -> Self;`

The number of elements (of type `Self`) in the list parameter of the signature `join` is equal to the number of the incoming transitions to the join node. The function `join` returns one entity of the species. It is used to group the data of several species entities within a single entity.

Finally, to synchronize all the incoming transitions to the join node, its outgoing transition must be conditioned by: $((\text{state}(x_1) = \llbracket Join \rrbracket) \wedge \dots \wedge (\text{state}(x_n) = \llbracket Join \rrbracket))$, where $x_i, i : 1..n$ belong to the incoming entities to the join node. This synchronization is ensured by a recursive function `states` (see Table 8).

Table 8: General Transformation of Join Nodes

<p>UML:</p>  <p>$\left(\begin{array}{l} \text{Join JoinNode} \\ \dots \\ \text{transition } tr_1(\text{Join}[\llbracket true \rrbracket] \dots) \\ \dots \end{array} \right)$</p> <p>FoCaLiZe</p> <pre> type nodes = $\llbracket Join \rrbracket$... ;; type transitions = $\llbracket tr_1 \rrbracket$... ;; species c_n = ... signature state: Self -> nodes ; signature join : list(Self) -> Self ; let aDJN(t:transitions, le:list(Self)):Self= match t with $\llbracket Join \rrbracket$ -> if (let rec states (l:list(Self)):bool = match l with [] -> true x::r -> (state(x) = $\llbracket Join \rrbracket$) && (states(r)) in states(le)) then join(le) else focalize_error("ERROR") _ -> focalize_error("ERROR") ; end;; </pre>
--

The function `states` checks that all the elements of the list `le` are incoming entities to the join node (`Join`).

To sum up, a fork node generates as many copies of the result object as the outgoing transitions (paths) using the function `ADFN` (its corresponding FoCaLiZe function is `aDFN`). The outgoing transition actions will be executed in parallel. Each outgoing transition uses one of the generated copies. Then, the following join node will group all of them, using the function `ADJN` (its corresponding FoCaLiZe function is `aDJN`). Hence, the function `ADJN` synchronizes and groups

the data of several objects (entities, on FoCaLiZe side) within a single object (entity on FoCaLiZe side). This synchronization of all the outgoing actions from the fork node is ensured by the recursive function `states` defined in Table 8. It uses the condition:

$$((\text{state}(x_1) = \llbracket \text{Join} \rrbracket) \wedge \dots \wedge (\text{state}(x_n) = \llbracket \text{Join} \rrbracket)), \text{ where } : x_i, i : 1..n.$$

In other words, the join node will not be launched until all the outgoing transitions (actions) from the fork node have been handled and reached the join state together.

4.2.2 Transformation of action constraints

Action constraints (local pre and post-conditions, see Fig. 1, page 6) are transformed in a similar way to OCL pre/post-conditions.

We convert local pre and post-conditions together into a FoCaLiZe implication ($\llbracket \mathbb{L}_{pre} \rrbracket \Rightarrow \llbracket \mathbb{L}_{post} \rrbracket$) of the corresponding species:

UML:

```
action «localprecondition»  $\llbracket \mathbb{L}_{pre} \rrbracket$ 
    op( $p_1 : typeExp_1 \dots p_k : typeExp_k$ )
    «localpostcondition»  $\llbracket \mathbb{L}_{post} \rrbracket$ 
```

FoCaLiZe:

```
property pre_post_action :
  all e : Self,
  all  $p_1 : \llbracket typeExp_1 \rrbracket$  , ... , all  $p_k : \llbracket typeExp_k \rrbracket$  ,
   $\llbracket \mathbb{L}_{pre}(e) \rrbracket \rightarrow \llbracket \mathbb{L}_{post}(e') \rrbracket$  ;
```

Where `op` is the name of the operation invoked by the action `action`, $p_1 \dots p_k$ are the operation parameters and $typeExp_1 \dots typeExp_k$ their corresponding types.

$\llbracket \mathbb{L}_{pre}(e) \rrbracket$ is the transformation of the pre-condition applied to the entity e (before the action is executed) and $\llbracket \mathbb{L}_{post}(e') \rrbracket$ is the transformation of the post-condition applied to the entity e' resulting from the action execution.

Transformation Example:

We can now complete our species `Order` (see Code 5) by the transformation of the class `Order` activity diagram (see Fig. 2 and Fig. 3, page 7) and its action constraints.

Code 5: Transformation of the class `Order` activity diagram

```
open "basics" ;;
type transitions = | Tr1 | Tr2 | Tr3 | Tr4 | Tr5 | Tr6 | Tr7 | Tr8 | Tr9 | Tr_Fork1
                | Tr_Join1 | Tr_final;;
type nodes = | InitialNode | Receive_order | Decision1 | Fill_order | Fork1 | Ship_order
            | Pay_order | Join1 | Merge1 | Close_order | FinalNode;;
(* The species Order, derived from the class Order *)
species Order =
signature get_filled : Self -> bool;   signature get_shipped : Self -> bool;
signature get_paid : Self -> bool;
signature newOrder : bool -> bool -> bool -> Self;
signature rejectOrder : Self -> bool;   signature fillOrder : Self -> Self;
signature billOrder : Self -> Self;     signature shipOrder : Self -> Self;
signature payOrder : Self -> Self;      signature joinOrder : list(Self)-> Self ;
signature closeOrder : Self -> Self;   signature initOrder : Self -> Self ;
signature state : Self -> nodes;

(* Transformation of the class order activity diagram *)
let aD (t:transitions, e:Self): Self = match t with
```

```

| Tr1 -> if (state(e) = InitialNode) then e else focalize_error("ERROR")
| Tr2 -> if state(e)= Receive_order then initOrder(e) else focalize_error("ERROR")

(* Handling of the node Decision1 - Decision not satisfied *)
| Tr3 -> if ((state(e)= Decision1) && (~ rejectOrder(e) )) then e else focalize_error("ERROR")
| Tr4 -> if (state(e) = Fill_order) then fillOrder(e) else focalize_error("ERROR")
| Tr5 -> if (state(e) = Ship_order) then shipOrder(e) else focalize_error("ERROR")
| Tr6 -> if (state(e) = Pay_order) then payOrder(e) else focalize_error("ERROR")

(* Handling of the node Merge1 *)
| Tr7 -> if (state(e) = Merge1) then e else focalize_error("ERROR")

(* Handling of the node Decision1 - Decision satisfied *)
| Tr8 -> if ((state(e) = Decision1) && (rejectOrder(e))) then e else focalize_error("ERROR")
| Tr9 -> if (state(e) = Close_order) then closeOrder(e) else focalize_error("ERROR")
| _ -> focalize_error("ERROR");

(* Handling of Fork Nodes *)
let rec aDFN (t:transitions , e:Self):list(Self)= match t with
| Tr_Fork1 -> if state(e)= Fork1 then [e; e] else focalize_error("ERROR")
| _ -> focalize_error("ERROR") ;

(* Handling of Join Nodes *)
let rec aDJN (t:transitions ,
le:list(Self)):Self = match t with
| Tr_Join1 ->
if (let rec states (l:list(Self)):bool = match l with
| [] -> true
| x::r -> (state(x) = Join1) && (states(r)) in states(le) ) then joinOrder(le)
else focalize_error("ERROR")

| _ -> focalize_error("ERROR") ;

(* Transformation of payOrder pre/post-constraints *)
property payOrder_pre_post: all e:Self, get_filled(e) -> get_paid(payOrder(e)) ;
(* Transformation of the action pay_order constraints *)
property action_pay_order_property: all e:Self, get_filled(e) -> get_paid(aD(Tr6, e)) ;
end::;

```

The property `action_pay_order_property` represents the transformation of the action `pay_order` localprecondition and localpostcondition.

4.2.3 Transformation of activity partitions

An activity partition (See Fig. 4) is an activity group for actions that invoke more than one class and have some common characteristics.

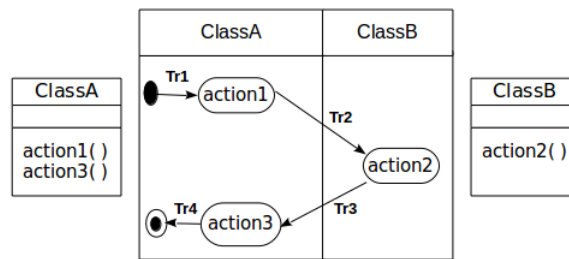


Figure 4: Activity Partition

To transform activity partitions, we use the same semantics and techniques as for activity diagrams (see Table 4). In addition, we use FoCaLiZe parameterized species (see section 2.2, page 4) to group the transformation of several classes within a single species. So, the activity partition presented in Fig. 4 is transformed into a function of the species `Partition` presented in

Code 6. The latter is parameterized by the species ClassA (derived from the class ClassA) and the species ClassB (derived from the class ClassB).

Code 6: Example of Activity Partition Transformation

```

type transitions = | Tr1 | Tr2 | Tr3 | Tr4 ;;
type nodes = | InitialNode | Action1 | Action2 | Action3 | FinalNode ;;

species ClassA =
  signature action1 : Self -> Self;
  signature action3 : Self -> Self;
  end;;
species ClassB =
  signature action2 : Self -> Self;
  end;;
species Partition(A is ClassA, B is ClassB)=
  representation = A * B ;
  signature state: Self -> nodes;

  let activityD (a:transitions, e:Self):Self= match a with
  | Tr1 -> if (state(e) = InitialNode) then e else focalize_error("ERROR")
  | Tr2 -> if (state(e) = Action1) then (A!action1(fst(e)), snd(e))
  else focalize_error("ERROR")
  | Tr3 -> if (state(e) = Action2) then (fst(e), B!action2(snd(e)))
  else focalize_error("ERROR")
  | Tr4 -> if (state(e) = Action3) then (A!action3(fst(e)), snd(e))
  else focalize_error("ERROR")
  | _ -> focalize_error("ERROR") ;
  end;;

```

5 Verification Approaches

Our goal is to provide a framework that automatically generates FoCaLiZe abstract specifications from UAD and OCL constraints. We can then use these specifications to check the consistency of the original model.

When we launch the transformation tool and the generated FoCaLiZe source is being compiled, several proof obligations are automatically generated. In particular, the proof of the derived properties from class invariants, pre and post-conditions and action constraints. At this stage, the automated theorem prover Zenon is (automatically) invoked to find proofs using function definitions and proof hints (see Fig. 5). If a proof fails, the FoCaLiZe compiler indicates the line of code responsible for the error. In this case, The FoCaLiZe developer analyses the source in order to correct and/or complete the UML model, and then restarts the development cycle.

According to the error message generated by the FoCaLiZe compiler, there are two main kinds of errors: either Zenon could not find a proof automatically, or there are inconsistencies in the original UML/OCL model. In the first case, developer interaction is needed to give appropriate hints to prove the properties, while in the second case one must go back to the original UML/OCL model to correct and/or complete it.

Note that it is fairly easy for the FoCaLiZe developer (who has a good understanding of the transformation) to determine the corresponding source elements in the UAD/OCL.

In addition to explicit properties (derived from OCL constraints), the FoCaLiZe compiler inserts implicit proof obligations when analyzing species. For example, if one function is redefined, all related proofs must be redone and any recursive definition must be done in parallel with its termination proof.

Although Zenon is the automatic theorem prover of FoCaLiZe, in a final step, each proof is transformed (automatically) to Coq scripts, and then rechecked. When we compile a FoCaLiZe

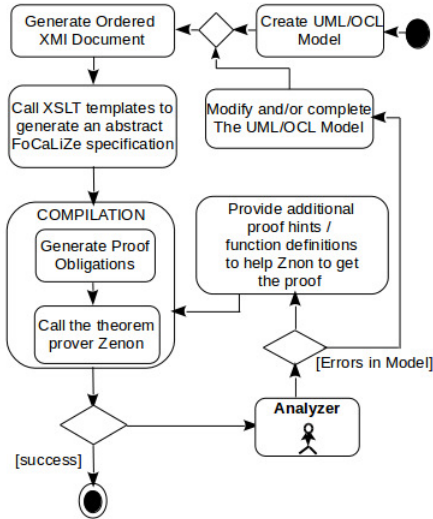


Figure 5: Transformation and Proof Framework

source (containing proof obligations), it is Zenon that will be invoked first. Then, all proofs will be automatically re-written into Coq terms for verification.

As a proof example, we present here an approach for **incoherency detection**. Using the proposed transformation, we obtain two kinds of properties from the original UML/OCL model:

- The properties derived from the OCL pre and post-conditions.
- The properties derived from action specifications (action local pre/post-conditions).

When a particular action of an activity diagram invokes one operation of the class and this operation is attached to an OCL pre and post-conditions, we obtain (in the FoCaLiZe specification) two properties referring to the same operation: the OCL pre/post-conditions and the action local pre/post-conditions. To detect contradictions, we assume that the property derived from the OCL pre/post-conditions (*pre-post_property*) is satisfied and then try to provide a proof (using FPL) of the property derived from the action specification (*action_property*) as follows:

`proof of action_property = by property pre-post_property ;`

If there is no contradiction between the two properties, Zenon will get the proof. Otherwise, the proof will be impossible.

To clarify this proof process, we complete the species `Order` with the following proof statements (Code 7):

Code 7: Successful proof of `action_pay_order_property`

```

species Order =
...
(* Transformation of payOrder pre/post-constraints *)
property payOrder_pre_post: all e:Self, get_filled(e) -> get_paid(payOrder(e)) ;

(* Transformation of the action pay_order constraints *)
property action_pay_order_property:

```

```

    all e:Self, get_filled(e) -> get_paid(aD(Tr6, e)) ;
proof of action_pay_order_property = by property payOrder_pre_post;
end;;

```

This example provides the proof of the property `action_pay_order_property`. It specifies the action `pay_order` of the class `Order` activity diagram.

The compilation of the above FoCaLiZe source ensures the correctness of the specification. No error has occurred, this means that the compilation, code generation and Coq verification were successful.

Imagine now that the UML user swaps (by mistake) the specification of the OCL pre/post-conditions of the operation `payOrder`. In this case, we will get the following FoCaLiZe source (Code 8):

Code 8: Unsuccessful proof of `action_pay_order_property`

```

species Order =
...
property payOrder_pre_post: all e:Self,
get_paid(e) -> get_filled(payOrder(e)) ;

property action_pay_order_property:
    all e:Self, get_filled(e) -> get_paid(aD(Tr6, e)) ;

proof of action_pay_order_property = by property payOrder_pre_post;
end;;

```

The compilation of this FoCaLiZe source returns the following message:

```

File "order.fcl", line 85, characters 27-57:
Zenon error: exhausted search space
without finding a proof
### proof failed

```

When Zenon cannot find a proof automatically, the FoCaLiZe compiler indicates the reason for the error, if any.

Finally, in order to validate the proposed transformation model, we have built firstly, a formal semantics for both UML and FoCaLiZe: Γ_U (for UML) and Γ_F (for FoCaLiZe). Then, a denotational semantics is defined based on these semantics. To be brief, only an overview about these semantics will be presented in the following paragraphs.

For a given class named cn , $\Gamma_U(cn) = (cn, V_{cn})$, where V_{cn} is the value of the class. A class value is a pair $(\Gamma_{cn}, body_{cn})$ composed of the local context of the class (Γ_{cn}) and the body of the class ($body_{cn}$), denoted by $body_{cn} = (\mathbb{A}_{cn}, \mathbb{O}_{cn}, \mathbb{C}_{cn}, \mathbb{AID}_{cn})$. It is composed of class attributes (\mathbb{A}_{cn}), class operations (\mathbb{O}_{cn}), class constraints (\mathbb{C}_{cn}) and the class activity diagram (\mathbb{AID}_{cn}):

- $\mathbb{A}_{cn} = \{(attr_n : typeExp)\}^*$, where $attr_n$ is an attribute name of the class cn and $typeExp$ its type.
- $\mathbb{O}_{cn} = \{(op_n : opType)\}^*$ where op_n is an operation name of the class cn and $opType$ its parameters and returned types.
- $\mathbb{C}_{cn} = \{(const_n : oclExp)\}^*$ where $const_n$ is a constraint name of the class cn and $oclExp$ its first order expression.

- $\mathbb{A}\mathbb{D}_{cn} = \{\mathbb{N}\mathbb{D}_{cn}\} \cup \{\mathbb{T}\mathbb{R}_{cn}\}$ where $\mathbb{N}\mathbb{D}_{cn}$ is a set of nodes and $\mathbb{T}\mathbb{R}_{cn}$ is a set of transitions.

The local context of the class cn (Γ_{cn}) is the list of its class parameters (\mathbb{P}_{cn}) and the list of its class dependencies (\mathbb{D}_{cn}): $\Gamma_{cn} = \mathbb{P}_{cn} \cup \mathbb{D}_{cn}$ where,

$\mathbb{P}_{cn} = \{(fp_n, typeExp) / fp_n \in \mathbb{P}_{cn}\}$ and $\mathbb{D}_{cn} = \{cn_i / cn_i \in \mathbb{D}_{cn}\}$

The semantics of the class activity diagram ($\mathbb{A}\mathbb{D}_{cn}$) is detailed in section 4.2.

Using the above definitions, we define **by induction**, the general semantics Γ_U of a UML/OCL model \mathbb{M} with multiple inheritance relationships, formal parameters, dependencies list, OCL constraints and activity diagrams.

In a symmetrical way, for a given species named sn , $\Gamma_F(sn) = (sn, V_{sn})$, where V_{sn} is the value of the species. A species value is a pair $(\Gamma_{sn}, body_{sn})$ composed of the local context of the species Γ_{sn} and its body:

$body_{sn} = (Sig_{sn}, Let_{sn}, Prop_{sn}, Proof_{sn})$, with

- $Sig_{sn} = \{(funName : funSig)\}^*$ where $funName$ is a function name and $funSig$ its signature.
- $Let_{sn} = \{(funName : funSig)\}^*$ where $funName$ is a function name and $funSig$ its signature.
- $Prop_{sn} = \{(propName : propSpec)\}^*$ where $propName$ is the name of a property and $propSpec$ its logical statement.
- $Proof_{sn} = \{(thmName : thmSpec)\}^*$ where $thmName$ is a theorem name and $thmSpec$ its logical statement.

During the transformation of the class cn to the species sn , the typing context of both UML/OCL ($\Gamma_U(cn)$) and FoCaLiZe ($\Gamma_F(sn)$) are enriched progressively, using semantics functions (inspired by the proposed transformation rules). Each typed element in $\Gamma_U(cn)$ will get a counterpart in $\Gamma_F(sn)$. By definition (of the semantics functions), each two paired elements in $\Gamma_U(cn)$ and $\Gamma_F(sn)$ having exactly the same semantics. So, we could prove the total correspondence between the semantics of the class cn and the semantics of the species sn .

6 Implementation

The UAD/OCL transformation is part of a wider FoCaLiZe project and hence needs to integrate with the pre-existing infrastructure. Therefore Eclipse based transformation tools such as QVT, ATL and ETL are not available within our current tool platform.

On the other hand, the XSLT¹ [23] processor is available in most programming environments, allows us to transform an XML document into various formats (XML, HTML, PDF, Text, etc.) and enables any change in its structure. In particular, we need to reorder the tags (elements of the XMI document) before performing the transformation. Finally, the combination of XMI and XSLT satisfies most conditions on tool independence and avoids (solves) the problem of model representation.

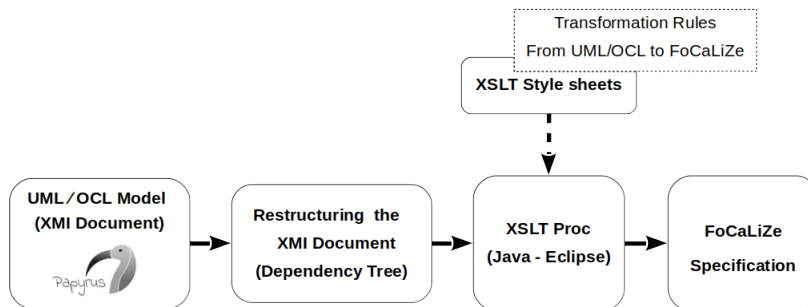


Figure 6: Systematic transformation of UML models into FoCaLiZe

Therefore, we have developed an XSLT stylesheet specifying the transformation rules from a UML model expressed in the XMI interchange format (generated by the **Papyrus** graphical tool) into FoCaLiZe (see Fig. 6).

The output is the corresponding FoCaLiZe source file, which can be directly read by the FoCaLiZe compiler (see Fig. 6). Additional information about the transformation tool (**UML2FOC**) are now available on-site², where instructions for installation and use are detailed.

7 Related Work

Some works deal with the transformation from UAD to the B method based tools. For example, [24] proposes the translation of activity diagrams into Event B, in order to verify workflow properties of distributed and parallel applications with the B prover. In a second work [25], a meta-model transformation from UAD to Event-B is proposed in order to verify functional properties of workflow models. A recent contribution aims to automate the generation of B language specification, starting from UML structural and behavioral diagrams and using graph grammar rules [26].

For comparison purposes, we studied the above transformation of UML/OCL models into Event-B method (which is the closest formal method to FoCaLiZe). We noticed that the navigation of OCL constraints via associations is not naturally formalized: each abstract machine (the main brick in a B method project) cannot access the properties of the other abstract machines. In addition, the transformed OCL constraints are not exploited to check the consistency of activity diagram transitions. In our transformation, we addressed the above limits using the FoCaLiZe parameterization (a client-supplier relationship) and late binding mechanisms. The latter allows a safe use (even at the abstract level) of all the functions and properties that are specified in the supplier species in order to develop functions and properties of the client species.

The transformation of an activity diagram (of a given class) into B ignores the cases where the class is created by multiple inheritance or from a UML template via a binding relationship. Using our proposal, the transformation of an activity diagram is always handled in a similar way, even if the class is created using the above UML features.

¹A usable language for the transformation of XML documents, recommended by the World Wide Web Consortium (W3C).

²<http://www.univ-eloued.dz/uml2foc/>

The Alloy tool is another formal method which has been used in the framework Alloy4SPV [27] for the verification of software process. The proposed framework uses a subset of UML2 Activity Diagrams as a process modeling language. However, class structures are isolated from activity diagrams.

Another approach consists of translating a UML activity diagram into CPN (Colored Petri Nets) specifications [28, 4] to attribute a formal semantics for UML activity diagrams and verify their properties. Recent contributions focus on the verification of SysML activity diagrams through their translation into Recursive RECATNets [29] and into Modular Petri Nets [30]. Yet here, neither OCL, action constraints nor activity partitions are considered. Another proposal using Colored Petri Nets [31] consists in the modeling of dynamic process aspects with CPN transformed from UAD.

A formal transformation of UAD based on rewriting logic and Graph Transformation [32, 33] is also proposed using the Maude system and its Real Time Checker.

The basic building blocks of UML 2.5 Activity Diagrams and their structural semantics have been formalized using Z Schemas [34].

Finally in [35], a shallow embedding and verification of a simple assembly language with procedures for arithmetic overflow. It uses a safety logic written as HOL (High Order Logic) through the proof assistant Isabelle/HOL [36]. Note that the language in question here is a very simple and limited language, without any high design and architectural features such as those advised by UML.

8 Conclusion and Perspectives

In this paper, we have proposed a functional semantics for UAD and then implemented a model transformation with FoCaLiZe. The proposed formal modeling supports the following points:

- Communication between a class structure and its activity diagram.
- OCL constraint: invariants and pre/post-conditions.
- Action constraints.
- Activity partitions.
- Formal implementation of control nodes (**DecisionNode**, **MergeNode**, **ForkNode** and **JoinNode**) using parallelism and choice statements in FoCaLiZe.

In addition, the high level design and architectural features of FoCaLiZe significantly reduces the gaps between UML/OCL and formal methods. The generated formal specification reflects perfectly both structural and behavioral aspects of the original UML/OCL model.

Moreover, using the automatic theorem prover Zenon, it is possible to prove the derived theorems and detect contradictions and deadlocks. Zenon indicates code portions responsible for errors, which leads developers to improve, complete and correct their original UML/OCL models.

In this paper we only focused on simple UML activity diagrams where activities are simple actions and behavioral expressions are limited to class operation calls. In future work, we will deal with additional UML2 activity diagram features such as complete activities with parameters and object nodes (pin and central buffer).

On the verification side, we will use the proposed transformation to deal with the verification of additional properties such as deadlock-freeness and the availability of system services.

References

- [1] OMG, UML : Unified Modeling Language, version 2.5, available at:<http://www.omg.org/spec/UML/2.5/PDF> (March 2015).
- [2] OMG, OCL : Object Constraint Language 2.4, available at: <http://www.omg.org/spec/{OCL}> (Jan. 2014).
- [3] D. Sarkar, M. Bhalla, S. M. Singal, Enhancing unified process workflows using uml, in: 2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence, IEEE, 2017, pp. 788–792.
- [4] J. Czopik, M. A. Košinár, J. Štolfa, S. Štolfa, Formalization of software process using intuitive mapping of UML activity diagram to CPN, in: Proceedings of the Fifth International Conference on Innovations in Bio-Inspired Computing and Applications IBICA, Springer, 2014, pp. 365–374.
- [5] R. Grønmo, I. Solheim, Towards Modeling Web Service Composition in UML., Web Services: Modeling, Architecture and Infrastructure, WSMAI 2004 4 (2004) 72–86.
- [6] W. M. P. V. der Aalst, Workflow verification: Finding control-flow errors using petri-net-based techniques, in: Proceedings of the International Business Process Management, Models, Techniques, and Empirical Studies, Vol. 1806, Springer, 2000, pp. 161–183.
- [7] J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 2005.
- [8] D. Jackson, Software Abstractions: Logic, Language and Analysis, MIT press, 2012.
- [9] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.
- [10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (Eds.), All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Vol. 4350 of Lecture Notes in Computer Science, Springer, 2007. doi:10.1007/978-3-540-71999-1.
- [11] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, Electronic Notes in Theoretical Computer Science 71 (2004) 162–187.
- [12] A. Messaoud, B.-Y. Choukri-Bey, R. Renaud, Generating FoCaLiZe Specifications from UML Models, in: Proceedings of the International Conference on Advanced Aspects of Software Engineering, ICAASE 2014, Constantine Algeria, 2014, pp. 157–164.
- [13] A. Messaoud, B.-Y. Choukri-Bey, R. Renaud, Modeling UML Template Classes with FoCaLiZe, in: The International Conference on Integrated Formal Methods, IFM2014, Bertinoro, Italy, Vol. 8739, Springer, 2014, pp. 87–102.
- [14] M. Abbas, C. Ben-Yelles, R. Rioboo, Formalizing UML/OCL structural features with focalize, Soft Comput. 24 (6) (2020) 4149–4164. doi:10.1007/s00500-019-04181-2.
URL <https://doi.org/10.1007/s00500-019-04181-2>
- [15] A. Messaoud, Using FoCaLiZe to check OCL constraints on UML classes, in: Proceedings of the International Conference on Information Technology for Organization Development, IT4OD 2014, Tebessa Algeria, 2014, pp. 31–38.
- [16] A. Messaoud, B.-Y. Choukri-Bey, R. Renaud, Modelling UML state machines with focalize, IJICT 13 (1) (2018) 34–54. doi:10.1504/IJICT.2018.10010449.
- [17] H. Thérèse, P. Francois, W. Pierre, D. Damien, FoCaLiZe : Tutorial and Reference Manual, version 0.9.2, CNAM-INRIA-LIP6, available at: <http://focalize.inria.fr> (2018).
- [18] P. Ayrault, H. Thérèse, P. François, Development life-cycle of critical software under focal, Electr. Notes Theor. Comput. Sci. 243 (2009) 15–31. doi:10.1016/j.entcs.2009.07.003.
- [19] M. Abbas, M. Beggas, A. Boucherit, Formalizing and verifying uml activity diagrams, in: International Conference on Model and Data Engineering, Toulouse, France, Springer, 2019, pp. 49–63.
- [20] R. Bonichon, D. Delahaye, D. Doligez, Zenon: An extensible automated theorem prover producing checkable proofs, in: Logic for Programming, Artificial Intelligence and Reasoning, LPAR, Yerevan, Armenia, Vol. 4790, Springer, 2007, pp. 151–165.
- [21] Coq, The Coq Proof Assistant, Tutorial and Reference Manual, Version 8.5, INRIA – LIP – LRI – LIX – PPS, Distribution available at: <http://coq.inria.fr/> (2016).
- [22] D. Delahaye, J. Étienne, V. Donzeau-Gouge, Producing UML Models from Focal Specifications: An Application to Airport Security Regulations, in: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, 2008, pp. 121–124.
- [23] W3C, XSL Transformations (XSLT) Version 3.0, W3C Recommendation, Oct. 2014, available at: <http://www.w3.org/TR/2014/wd-xslt-30-20141002/> (2015).
- [24] A. B. Younes, L. J. B. Ayed, Using UML activity diagrams and event B for distributed and parallel applications, in: Computer Software and Applications Conference. COMPSAC 2007, 31st Annual International, Vol. 1, IEEE, 2007, pp. 163–170.
- [25] A. B. Younes, Y. B. Hlaoui, L. J. B. Ayed, A Meta-model Transformation from UML Activity Diagrams to Event-B Models, in: Computer Software and Applications Conference Workshops (COMPSACW), IEEE, 2014, pp. 740–745.
- [26] S. Rehab, A. Chaoui, Tgg-based process for automating the transformation of uml models towards b specifications, International Journal of Computer Aided Engineering and Technology 7 (3) (2015) 378–400.

- [27] Y. Laurent, R. Bendraou, S. Baair, M.-P. Gervais, Alloy4spv: A formal framework for software process verification, in: *European Conference on Modelling Foundations and Applications*, Vol. 8569, Springer, 2014, pp. 83–100.
- [28] N. Maneerat, W. Vatanawood, Translation uml activity diagram into colored petri net with inscription, in: *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, IEEE, 2016, pp. 1–6.
- [29] M. Rahim, A. Kheldoun, M. Boukala-Ioualalen, A. Hammad, Recursive ECATNets-based approach for formally verifying System Modelling Language activity diagrams, *IET Software* 9 (5) (2015) 119–128.
- [30] M. Rahim, A. Hammad, M. Boukala-Ioualalen, Towards the formal verification of sysml specifications: Translation of activity diagrams into modular petri nets, in: *Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence (ACIT-CSI)*, 2015 3rd International Conference on, IEEE, 2015, pp. 509–516.
- [31] J. Czopik, M. A. Košinár, J. Štolfa, S. Štolfa, Addition of static aspects to the intuitive mapping of UML activity diagram to CPN, in: *Afro-European Conference for Industrial Advancement*, Springer, 2015, pp. 77–86.
- [32] E. Kerkouche, K. Khalfaoui, A. Chaoui, A. Aldahoud, UML Activity Diagrams and Maude Integrated Modeling and Analysis Approach Using Graph Transformation, in: *Proceedings of the 7th International Conference on Information Technology (ICIT 2015) doi*, Vol. 10, 2015.
- [33] E. Kerkouche, K. Khalfaoui, A. Chaoui, A. Aldahoud, UML Activity Diagrams and Maude Integrated Modeling and Analysis Approach Using Graph Transformation, in: *Proceedings of ICIT 2015 The 7th International Conference on Information Technology*, Amman, Jordan, 2015, pp. 515–521.
- [34] M. Jamal, N. A. Zafar, Formalizing structural semantics of UML 2.5 activity diagram in Z notation, in: *Open Source Systems & Technologies (ICOSST)*, 2016 International Conference on, Lahore, Pakistan, IEEE, 2016, pp. 66–71.
- [35] M. Wildmoser, T. Nipkow, Certifying machine code safety: Shallow versus deep embedding, in: *International Conference on Theorem Proving in Higher Order Logics*, Springer, 2004, pp. 305–320.
- [36] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-order Logic, Vol. 2283, 2002.