Studies in Engineering and Exact Sciences
ISSN: 2764-0981

A maintainable and iterative development approach of critical systems with FoCaLiZe

Uma abordagem de desenvolvimento sustentável e iterativo para sistemas críticos com FoCaLiZe

DOI: 10.54021/seesv5n1-092

Recebimento dos originais: 16/04/2024
Aceitação para publicação: 03/05/2024

Fatima Haloua
Master in Management of Information Systems
Institution: LIAP Laboratory, University of El Oued
Address: PO Box 789, El Oued, Algeria
E-mail: haloua-fatima@univ-eloued.dz

Messaoud Abbas
PhD in Computer Science
Institution: LIAP Laboratory, University of El Oued
Address: PO Box 789, El Oued, Algeria
E-mail: messaoud-abbas@univ-eloued.dz

Ammar Boucherit
PhD in Computer Science
Institution: LIAP Laboratory, University of El Oued
Address: PO Box 789, El Oued, Algeria
E-mail: ammar-boucherit@univ-eloued.dz

Hessa Alfraihi
PhD in Computer Science
Institution: Department of Information Systems, College of Computer and Information Sciences, Princess Nourah bint Abdulrahman University
Address: P.O. Box 84428, Riyadh, Saudi Arabia
E-mail: haalfraihi@pnu.edu.sa

ABSTRACT

The life-cycle development of critical systems follows mainly a V-cycle process overlapped with proofs and/or verification techniques. At the first development stage, a critical system must be, precisely, and as completely as possible described. Thus, it is recommended to use a graphical modeling language standard such as the Unified Modeling Language (UML) and the Object Constraint Language (OCL), to which both customers and developers could contribute. Then, the graphical design with its formal specification (requirements) is mapped into a complete formal development environment, using an MDE (Model-driven engineering) approach. The formal environment has to assess the development until the generation of a secure code. In this context, we propose a life-cycle development approach that combines UML/OCL and the FoCaLiZe formal

environment for the secure development of critical systems. The proposed approach ensures essential development constraints such as strict boundaries and traceability between development phases. It also ensures a secure maintenance phase using partial transformation techniques from UML/OCL to FoCaLiZe. The latter enhances the transformation from UML/OCL into FoCaLiZe to support incremental development and assist developers in correcting modeling errors. The proposed approach is mostly dedicated to secure and maintainable lifecycle development processes of critical systems. The formal environment assesses the development process until the generation of a secure code. To illustrate the proposed approach, we present the stages of development of a constrained system controlling military objects. The development of this system uses the iterative, incremental, and maintainability stages as described by the development approach, until the generation of secure code.

Keywords: software engineering, development lifecycles, model verification, formal methods, UML/OCL, FoCaLiZe.

RESUMO

O desenvolvimento do ciclo de vida de sistemas críticos segue principalmente um processo de ciclo em V, sobreposto com técnicas de prova e/ou verificação. Na primeira etapa de desenvolvimento, um sistema crítico deve ser, precisamente e tão completamente quanto possível, descrito. Assim, é recomendado usar uma linguagem de modelagem gráfica padrão, como a Linguagem de Modelagem Unificada (UML) e a Linguagem de Restrição de Objetos (OCL), à qual tanto os clientes quanto os desenvolvedores podem contribuir. Em seguida, o design gráfico com sua especificação formal (requisitos) é mapeado para um ambiente de desenvolvimento formal completo, usando uma abordagem de Engenharia Orientada a Modelos (MDE). O ambiente formal deve avaliar o desenvolvimento até a geração de um código seguro. Nesse contexto, propomos uma abordagem de desenvolvimento do ciclo de vida que combina UML/OCL e o ambiente formal FoCaLiZe para o desenvolvimento seguro de sistemas críticos. A abordagem proposta garante restrições de desenvolvimento essenciais, como limites rígidos e rastreabilidade entre as fases de desenvolvimento. Também garante uma fase de manutenção segura usando técnicas de transformação parcial de UML/OCL para FoCaLiZe. Esta última melhora a transformação de UML/OCL para FoCaLiZe para suportar o desenvolvimento incremental e ajudar os desenvolvedores a corrigir erros de modelagem. A abordagem proposta é principalmente dedicada a processos de desenvolvimento de ciclo de vida seguros e mantíveis de sistemas críticos. O ambiente formal avalia o processo de desenvolvimento até a geração de um código seguro. Para ilustrar a abordagem proposta, apresentamos as etapas de desenvolvimento de um sistema restrito que controla objetos militares. O desenvolvimento desse sistema usa as etapas iterativas, incrementais e de manutenção conforme descrito pela abordagem de desenvolvimento, até a geração de código seguro.

Palavras-chave: engenharia de software, ciclos de vida do desenvolvimento, verificação de modelo, métodos formais, UML/OCL, FoCaLiZe.

1 INTRODUCTION

The typical phases in a formal (critical) development process include requirement specification, architecture design, implementation, testing, and maintenance. The development lifecycle of critical systems is of particular interest in verification throughout all phases of development. The system requirements must be satisfied by the software components.

Standards such as IEC-61508 [19] and common criteria for security evaluation [49] provide verification requirements for each stage of the lifecycle. In this context, formal methods are highly recommended to specify, code and verify such systems. Formal methods, based on proof or simulation, typically offer an unambiguous language for expressing systems reasoning on properties (requirements). On the other hand, the development lifecycle of critical systems should enable developers to connect more with customers and authorize new requirement integration. In addition, it should support an iterative process to correct errors that may have occurred at each development phase. Otherwise, the development of critical systems should be handled in several iterations. Each iteration should correct the previously identified errors and may incorporate new features into the system.

Usually, developers employ a graphical modelling tool, such as UML [65], during the initial stages of development. The graphical tool is recommended to serve as an intermediary between non-specialized domain experts and specialized users (developers) [73]. However, UML notations lack formal semantics for rigorous system specification. Therefore, in conjunction with UML, we need to use a formal specification language with well-defined syntax to enable precise specification of functional requirements.

Such languages ensure rigorous semantics for requirements specification. The Object Constraint Language (OCL) [64] is commonly used alongside UML for this purpose.

This article proposes a lifecycle process for the development of critical systems that addresses the following challenges:

- Tests and/or verification at each development phase.
- Iterative and incremental development that encourages new requirement integration.

We have chosen FoCaLiZe [46], a formal proof-based environment that covers most of the lifecycle development phases to address these challenges. FoCaLiZe provides mechanisms for the abstract design and specification of models. It also enables incremental development using object-oriented features. Furthermore, FoCaLiZe integrates a language for requirements specification, a functional programming language for code generation, and proof/test techniques for verification and simulation. The FoCaLiZe theorem prover is called Zenon [14], which automatically fulfills the verification of the system obligation, facilitating user intervention. Zenon generates (automatically) a [27] code that the Coq theorem prover checks. Coq acts as an assessor of both proofs and model consistencies.

Thus, this paper proposes a partial transformation approach from UML/OCL to FoCaLiZe to ensure iterative and incremental development with the aim to enhance the maintenance phase. The proposed approach uses the mapping of UML/OCL elements developed in previous works: static aspects (of the UML/OCL models) were mapped in [1–3], and dynamic aspects were mapped in [4, 6]. An overview of these mappings will be presented in the proposed approach section.

To illustrate this proposal, we present the development stages of a critical system that controls military objects. The first iteration of the system considers military objects on the surface in a two-dimensional Cartesian plane. Then, a new increment is introduced to consider military objects in space, using a three-dimensional plane. The ultimate goal of this paper is to introduce an innovative methodological approach for the incremental and iterative development of critical systems, by combining the formal methods of UML/OCL and FoCaLiZe. Thus, we aim to provide a comprehensive framework to address the specific challenges encountered in the development of critical systems, with an emphasis on rigorous verification at every stage of the development process.

The background of this study is presented in Sections 2.1 and 2.2.The first one presents the UML/OCL specification and the design of the military object system, while the second describes the FoCaLiZe architecture and design features. Then, Section 3 presents the proposed approach for a maintainable and iterative development lifecycle of critical systems. Finally, Section 4 highlights some related work, before concluding in Section 5.

2 BACKGROUNDS

2.1 UML/OCL MODEL

Software engineers commonly adopt UML/OCL as a modelling standard during the development lifecycles, such as in Rational Unified Process (RUP), Agile Unified Process (AUP) and OpenUP [10,39].

A UML model is a set of diag rams that describe the static and behavioral aspects of the system. Software engineers employ UML to design, visualize, and document software systems. On the other hand, OCL is a formal declarative language that provides constraints on UML model components.

We use UML/OCL to specify requirements and design critical systems in the first development stage. The class diagram describes the design of the system components. It is a class set with attributes and operations connected with relationships: inheritance and multiple inheritances (with methods' redefinition and late binding), dependency, and associations between classes.

Customer requirements are gathered and specified using OCL constraints. The latter are either invariants of classes or pre/post conditions for class operations.

Most UML/OCL tools generate an XMI document, XML Metadata Interchange, as a textual description of the UML/OCL model elements [66]. The XMI document is the only source for any eventual transformation of UML/OCL models into other formalisms. For the description of OCL constraints, we follow the standard OCL language syntax [64].

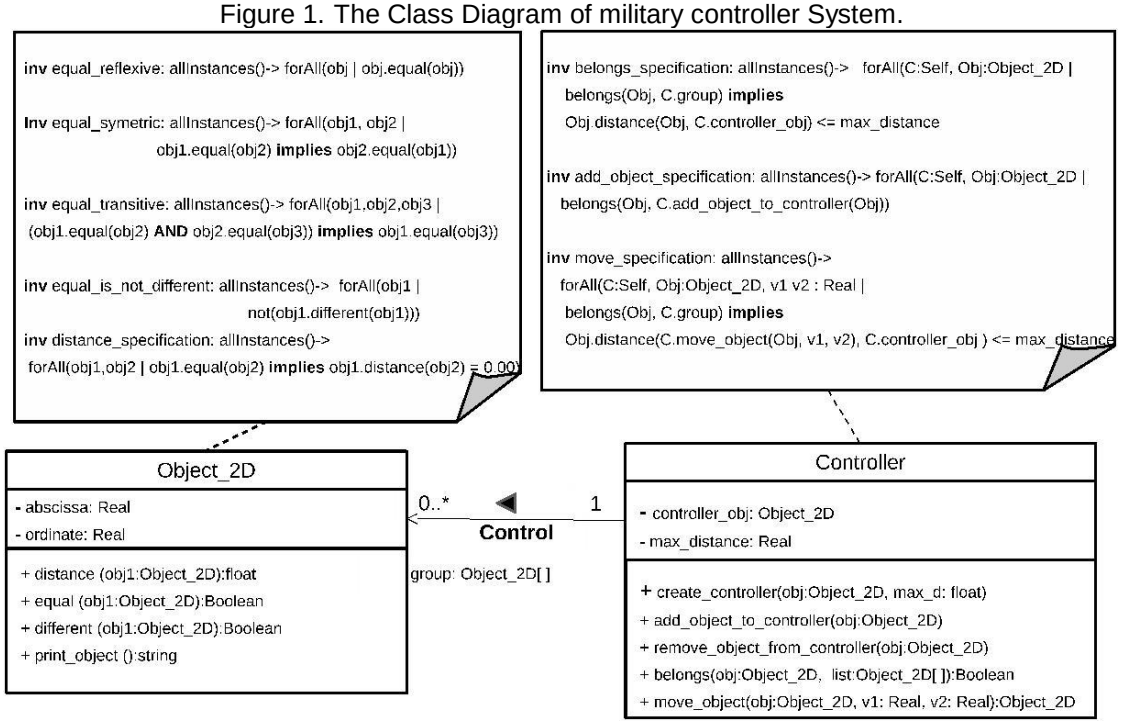
The developed case study is a military system that controls and locates military objects on the surface.

The initial requirements of the system consist of controlling and locating military objects on the surface, 2D objects. Therefore, the location of military objects is modelled on a two-dimensional Cartesian plane, with two axes: the x-axis and y-axis. Each object has two coordinates: an abscissa and an ordinate. An object is designed as a controller for each group of neighbouring objects. A controller manages and controls a group of military objects. Each military object Obj that belongs to a controller C should not exceed a maximum distance (max_distance) from the controller. A controller, regarding its group of controlled objects, should be able to:

- get and update the abscissa and ordinate values of each controlled military object;
- calculate the distance between two controlled objects, including the controller itself;
- move an object from one position to another by providing updated values of d-abscissa and d-ordinate of the desired position;
- add an object to the controller group;
- remove an object from the controller group;
- decide, at any moment, whether an object Obj belongs to the controller or not;
- print military object information.

Figure 1 shows the class diagram of the aforementioned system. The class Object_2D models military objects on the surface. The distance method calculates the distance between two given class instances. The method equal and the method different are used to decide the equality between objects. The class Controller has two attributes: the controller_obj (the controller itself) and the max_distance (the maximum distance between the controller and objects). The unidirectional association between the classes Controller and Object_2D models the group of controlled objects. A new controller is created with an empty group of objects. Then, we add and remove objects using methods add_object_to_controller and remove_object_from_controller. When a new object is added to a controller, it means that it satisfies the constraint against the maximum distance. Similarly, moving an object from one position to another should satisfy the specified constraints. These constraints are ensured through the OCL invariants: belongs_specification, add_object_specification and move_specification.

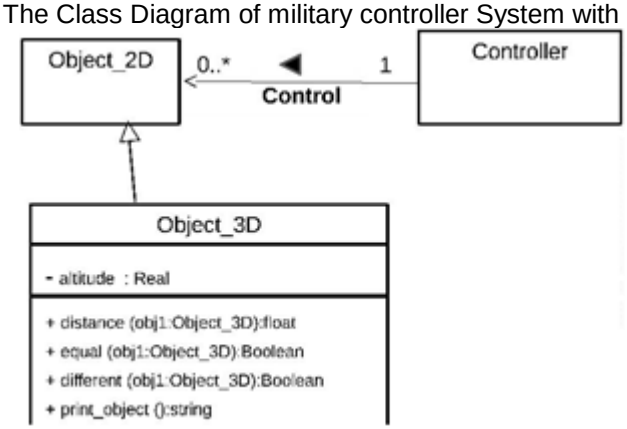
Figure 1. The Class Diagram of military controller System.



Source: Authors.

New requirements are introduced after the client company implements, deploys, and makes this system usable. The client company needs to consider objects in a three- dimensional space (not only on the surface). Objects in the space, Object-3D, are located with an altitude using a third axis (z-axis) in addition to abscissa and ordinate. Therefore, the company should amend the original system at the lowest possible cost. The modified system is specified using the class diagram presented in Fig. 2. The class Object-3D inherits the class Object_2D. The only newly added attribute is a three-dimensional military object's altitude. However, due to this rational change between two-dimensional and three-dimensional objects, most inherited methods from the class Object_2D should be redefined.

Figure 2. The Class Diagram of military controller System with increments



Source: Authors.

2.2 FOCALIZE AND LIFECYCLES

FoCaLiZe is a complete formal programming environment. FoCaLiZe follows object- oriented Paradigm (OOP) for system design and architecture. It uses a basic brick called a species (it resembles classes in Object Oriented Programming). A species is an abstraction of a collection of entities that share the same data type and use the same methods. In addition, the species may contain properties (requirement specification) that the entities of the species should satisfy. Properties are first order logical statements called theorems when proven. FoCaLiZe uses a proof-based approach. Thus, we cannot create an entity from one species unless it satisfies all the proclaimed requirements (properties). Table 1 presents the general syntax of a species.

Table 1. The Syntax of a Species

spec ::= **species** speciesName [(param [{ , param}∗])] = [**inherit** specDef [{ , specDef}∗] ;] {methods;}∗ **end**;;
param ::= ident **in** type | ident **is** specDef
specDef ::= speciesName |speciesName (param [{ , param}∗])
methods ::= rep | signature | let |property | theorem
rep ::= **representation** = type;
signature ::= **signature** functionName:functionTtype;
let ::= **let** [**rec**] functionName = functionBody;
property ::= **property** propertyName : propertySpecification ;
theorem ::= **theorem** propertyNname : propertySpecification **proof** = theoremProof;

Source: Authors.

The representation defines the data type of the species entities. The terms between parentheses after the species name are the formal parameters of the

species. They are either another species of the model or an entity from an existing species. Thus, one species may be parameterised by other species (at the abstract level) and/or with concrete entities generated from complete species.

The parameterisation is similar to the dependency relationship in OOP. When a species A is parameterised with another species B, it can use all members (representation, methods and properties) of B to develop its members at the abstract level.

After the keyword inherit, we find a list of species from which the current species inherits. As within OOP, a subspecies inherits all members of its superspecies and may define its own members.

Each method of the species may be defined abstractly or concretely. The representation of a species may not be defined (abstract) in the early stages of development. But it has to be concretely defined by a type expression in the last step. Similarly, a species method may be abstract, denoted by the keyword signature, and ultimately defined by a computational body, using the keyword let. Finally, species requirements are stated as logical properties using the keyword property and must be proven before using the species and generating entities from it. Thus, we can only create entities from complete species. In complete species, representations, methods, and properties are thoroughly defined. This distinction between abstract and implementation levels is to enable incremental development of species. We often initiate the system to develop with some connected abstract species (with signatures and properties). Then, we refine and enrich these species progressively until we obtain the complete species. To use a complete species, we generate a collection from it. The collection is a final user component that interfaces its original complete species.

Several collections may be generated from one complete species by substituting its formal parameters with actual ones.

Hence, for the development of this work, we have chosen the FoCaLiZe environment since it fulfils most lifecycle stages within a formal scope: formal requirement specification, formal modelling, formal coding (programming), formal verification throughout all development stages, documentation, and the generation of a secure executable (without risks):

- **Formal requirement specification**: For requirement specification, FoCaLiZe integrates a specification language that enables us to express requirements using first-order logic. These logical expressions rely on members (representation, methods, and properties) of the current species and also members of supplier species (species used as parameters).
- **Formal modelling:** For the design of system models, FoCaLiZe follows OOP techniques, such as inheritance, dependency (parameterisation), templates and template binding. These features are used in an abstract and formal context. Species properties, methods, and proofs could be safely propagated and used through these design features.
- **Formal coding (programming):** Methods implementation (definition) is achieved using a purely functional language inspired by ML expressions (let-binding, pattern matching, conditional, higher-order functions . . .). FoCaLiZe also enables recursive definitions through the keyword let rec. The functional programming paradigm is highly recommended in a formal development environment to avoid programming side effects from anomalies of imperative programming languages (with variables and assignments).
- **Formal verification:** To check customer requirements, which are properties within species, FoCaLiZe uses theorem provers. The backbone theorem prover of FoCaLiZe is Coq, but to make the proof tasks easier and more automatic, FoCaLiZe integrates the Zenon theorem prover. Thus, to prove one property, the FoCaLiZe developer targets proof hints (indications) to Zenon. When Zenon finds the proof, it automatically translates it to Coq to confirm and legalise the obtained proof. Proof hints are written using FPL. FPL is an integrated language to write proof hints. Zenon uses the proof hints to discharge the proof. Through FPL, a proof hint is a statement like: proof <theorem name>; to invite Zenon to find a proof automatically for one theorem. Developer interaction is required when Zenon fails to find a proof automatically. In this case, the proof hints are presented in several steps breaking down the main goal into many sub-goals. Each sub-goal is handled in specific steps. The final step should conclude with the primary objective of the theorem. In addition to FPL and theorem provers, FoCaLiZe

integrates a test tool, called FocalTest, for the validation of unprovable theorems, Zenon and Coq cannot prove them.

- **Documentation:** FoCaLiZe uses its integrated documentation tool FoCaLiZeDoc [6] to generate documentation. The documentation is an XML file that is automatically generated from many sources: the FoCaLiZe source, structured comments added by the programmer, the concrete syntax of FoCaLiZe, and the outcomes of type inference and dependency analysis. Thanks to some XSLT stylesheets, FoCaLiZeDoc may also generate HTML pages or LATEX files.
- **Secure Executable code:** The FoCaLiZe environment guides the developer to provide all definitions and proofs before using his system (species). At the ultimate step, when everything goes correctly, the FoCaLiZe compiler provides a safe (certified) executable that the final user can use without risk (all customer requirements have been satisfied).

3 THE PROPOSED APPROACH

This section describes the proposed partial transformation of UML/OCL models into FoCaLiZe. Then, on the basis of this transformation, it shows a maintainable and iterative development approach (of critical systems).

3.1 OVERVIEW OF THE PROPOSED APPROACH

Fig. 3 shows the main steps of the proposed lifecycle process. It integrates a UML/OCL tool, the full and partial transformation rules, and the FoCaLiZe environment. The approach involves a development team with UML/OCL users and FoCaLiZe experts. Thus, a formal development through this approach should follow the next steps:

The proposed development process starts with gathering and specifying all system requirements. Then, we use a UML2 tool equipped with the OCL constraints specification language to design the system. After that, we automatically generate the standard XMI document, an XML representation of the UML model. It is worth noting that the majority of UML/OCL tools offer this automatic generation. The next step involves reconfiguring the produced XMI document to ensure the proper dependencies between UML model elements

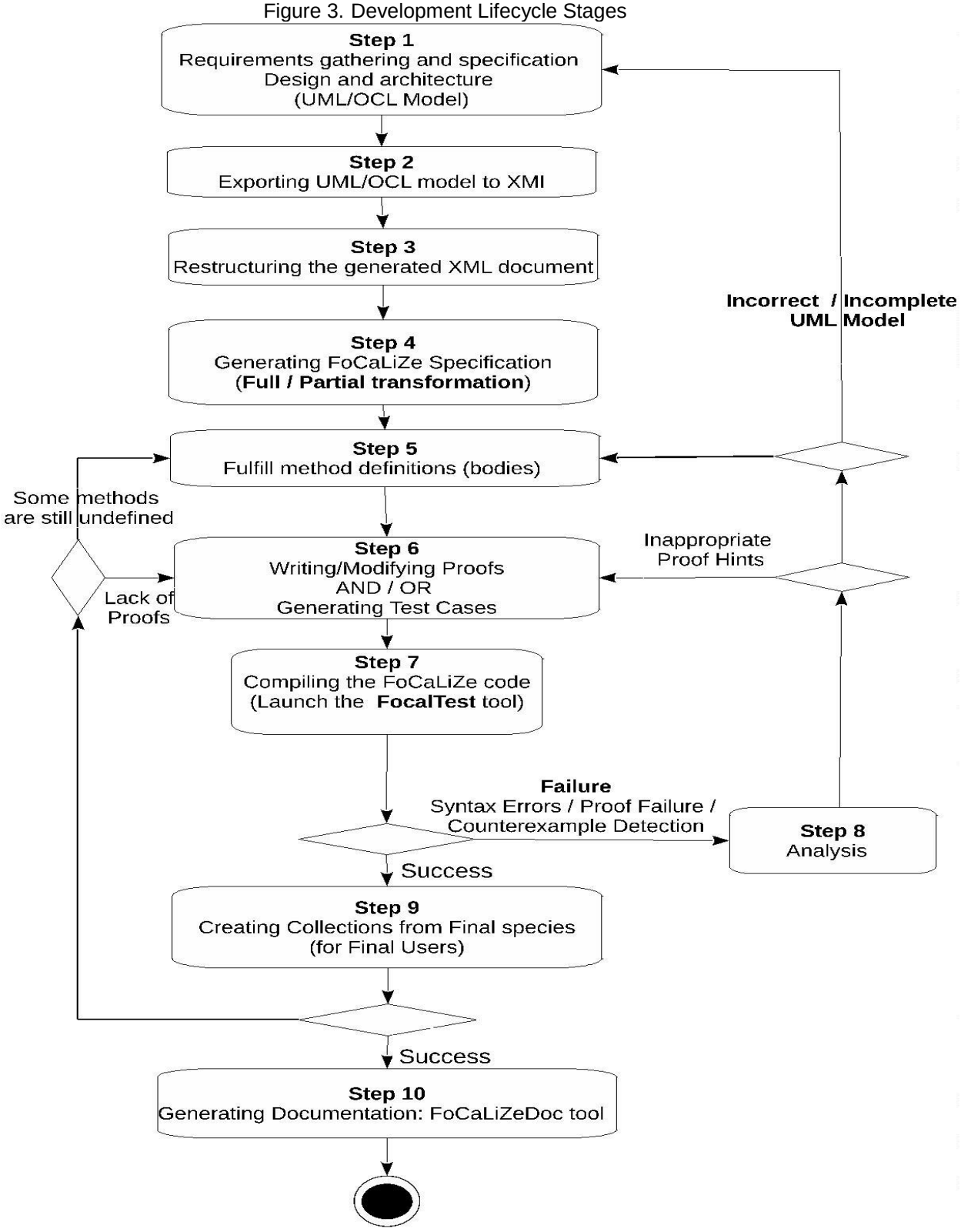
during transformation. From the UML/OCL model, an Abstract Code specification is automatically generated. The generated code is then completed by a FoCaLiZe developer, ensuring that all the system methods are defined by their computational bodies. We then manually integrate proofs, where we provide proof hints (tactics) for the derived properties. These hints will be used by Zenon (FoCaLiZe's automatic theorem prover) to fulfil proofs during code compilation. If Zenon succeeds, it produces a Coq code that will be verified by the Coq theorem prover. However, if the proof fails, the FoCaLiZe developer will need to identify the source of the error and adjust accordingly, whether it is an issue with the original UML/OCL model or improperly provided proof hints. This entire process can go through multiple iterations, ensuring that all errors are rectified. Through these iterations, new increments in the system can be added. Ultimately, Collections, the final components intended for users, are created from the final species, and documentation is generated from the FoCaLiZe source.

3.2 PARTIAL TRANSFORMATION

The partial transformation of a UML/OCL model into FoCaLiZe consists in transforming only part of the UML/OCL model into FoCaLiZe. The whole UML/OCL model should be transformed in the first iteration into FoCaLiZe. After that, we will need to transform only a part of the original UML/OCL for iteration and maintenance purposes. We should not, of course, clear all the old FoCaLiZe code generated in the first transformation round. In fact, the abstract FoCaLiZe code generated in the first transformation iteration will be manually affected and completed by a FoCaLiZe developer. Thus, we should not lose the efforts of the FoCaLiZe developer, and we have to update the FoCaLiZe code by transforming only the amended UML/OCL components.

We need to restart the development cycle from scratch in the maintenance phase. In particular, when we introduce new requirements, correct errors that were not discovered

Figure 3. Development Lifecycle Stages



Source: Authors.

During the design and architecture phase, or enhance the system with new functionalities. We should not erase the previously generated and amended code in all these cases.

When we talk about the UML class diagram, the partial transformation may concern classifier components such as classes or associations between classes, as it may concern class members, such as attributes, operations, or OCL constraints of classes.

In the following paragraphs, we describe an approach (algorithm) for the partial transformation of UML/OCL models into FoCaLiZe. This algorithm will be integrated within the general transformation approach of the UML / OCL models into FoCaLiZe and then used to perform new increments and maintenance phases; see Fig. 3.

To develop the partial transformation algorithm, we should first remind the following formal description of a UML class and its transformation:

$$[\textbf{public}][\ll \text{class} - \text{streotype} \gg] \textbf{ class } \text{cn } (P_{cn}) \textbf{ binds } T_{cn} \textbf{ depends } D_{cn} \textbf{ inherits } H_{cn} = \quad (1)$$

Notation:

In the following, we denote [[e]] the transformation of a UML/OCL element e into FoCaLiZe.
The above definition shows the general structure of a UML class cn with relationships to other classes. In this definition,
P_{cn} is the list of the formal parameters of the class cn;
T_{cn} is the list of substitutions of the formal parameters of the class cn with actual parameters;
H_{cn} is the list of classes from which the class cn inherits;
D_{cn} is the list of classes on which the class cn depends;
A_{cn} is the list of attributes of the class cn; and
O_{cn} is the list of the class cn operations.

The general transformation rule of the class cn into a species sn is presented as follows:

UML:

$$[\textbf{public}][\ll \text{class } - \text{stereotype} \gg] \textbf{ class } \text{cn } (P_{cn}) \textbf{ binds } T_{cn} \textbf{ depends } D_{cn} \textbf{ inherits } H_{cn} = A_{cn}\ O_{cn} \textbf{ end } ; ; \quad (2)$$

FoCaLiZe:

$$\text{species sn } (P_{sn}) = \text{inherit } [[H_{sn}]];\ \ [[A_{sn}]]\ [[O_{sn}]] \text{ end } ; ; \quad (3)$$

Where:

P_{sn} is the generated list of the species sn formal parameters,
H_{sn} is the generated list of species from which the species sn inherits,
Sig_{sn} is the generated list of signatures of the species sn. It includes getters functions derived from the class attributes and methods signatures derived from the class operations.
$Prop_{sn}$ is the generated list of properties of the species sn.
$Proof_{sn}$ is the list of the already discharged proofs within the species sn.

The transformation rules, from UML class diagram concepts into FoCaLiZe, have already been detailed in [2, 1–3]. First, we explain the main transformation rules needed for developing the proposed lifecycle. Then, we enhance these rules by the proposed partial transformation to ensure maintainability and interactivity. The essential transformation rules are:

- Each primitive type of UML has a corresponding FoCaLize type. For example, the UML type Integer corresponds to the FoCaLiZe type int:

Primitive Types :	[Integer]]	= int
	[Real]]	= float
	[[Boolean]]	= bood
	[String]]	= string
	[UnlimitedNatural]]	= unlimited_Nat

A general UML type t expression with multiplicity is transformed into the FoCaLiZe type List(t), which is a list of elements of the type t

The transformation rules, from UML class diagram concepts into FoCaLiZe, have already been detailed in [9, 9–11]. First, we explain the main transformation rules needed for developing the proposed lifecycle. Then, we enhance these rules by the proposed partial transformation to ensure maintainability and interactivity. The essential transformation rules are:

- Each primitive type of UML has a corresponding FoCaLize type. For example, the UML type Integer corresponds to the FoCaLiZe type int:

$$[\ \text{typeExp}\ [\text{mult}]]] = \begin{cases} [\text{typeExp}]] \text{ if mult} = 1..\ 1 \\ .\text{ist}\ ([[\text{typeExp}]]) \text{ if mult} \neq 1..\ 1 \end{cases}$$

- Each class attribute (instance variable) gives rise to a signature modeling its getter function in the corresponding species.

UML:

$$\text{attrNane} : \text{typeExp}\ [\text{mult}]; \tag{4}$$

FoCaLiZe:

$$\text{signature}\quad \text{get}_{\text{attrNane}} : \text{Self}\ \text{–>}\ [[\text{typeExp}\ [\text{mult}]]]; \tag{5}$$

- We transform class methods into species signatures (function interfaces). Each species signature starts with the type Self, representing the entity that invokes the function. Thus, the signature starts with Self, followed by the types of the function parameters, and ends with the newly created entity represented by the type Self. This general rule has two exceptions. The first one regards class constructors whose target signatures do not start with the keyword Self. The second exception regards operations (methods) that return a value. Their target signature ends with the corresponding type in FoCaLiZe.

UML:

$$\text{SN}\begin{pmatrix}\text{dir}_1\text{p}_1 : \text{type}_1[\text{mult}_1] \\ \ldots \\ \text{dir}_k\text{p}_k : \text{type}_k[\text{mult}_k]\end{pmatrix} : \text{Type}[\text{mult}]$$

FoCaLiZe:

$$\text{Signature N}: [\text{Self}]\ \text{–>}\ \left[\text{Type}_1[\text{mult}_1]\right]\text{–>}\quad \ldots\ \text{–>}\ \left[\text{Type}_k[\text{mult}_k]\right]\text{–>}\ [\text{Self}]; \tag{6}$$

N is the operation name, V its visibility and S its stereotype. The operation parameter directions are dir_i (in, by default or out). The p_i are the operation

parameter names and type_itheir types. The operation return type is Ty pe and has multiplicity mult.

- The primitive types of OCL, Integer, Real and String, are transformed into their equivalent primitive types of FoCaLiZe. We also transformed most OCL operations of the Integer, Real, and String sets into their corresponding operations on FoCaLiZe. Boolean formulas specified in OCL are straightforwardly transformed into Boolean expressions of FoCaLiZe. We transform the three types of OCL constraints (invariants, pre-conditions and post- conditions) that specify the constraints on UML classes into FoCaLiZe properties of the corresponding species. We transform each pre- and post-conditions of the same class operation into a FoCaLiZe implication (pre-condition ⇒ post-condition).

An OCL invariant E_{inv} of the class cn is converted into a FoCaLiZe property of the corresponding species:

OCL:

$$\text{context cn inv:} \quad \text{E}_{\text{inv}} \tag{7}$$

FoCaLiZe:

$$\text{property invldent: all e: Self, } [[\text{E}_{\text{inv}}]]; \tag{8}$$

An OCL pre- and post-conditions E_{pre} and E_{post} of an operation named N of the class cn are transformed together into a FoCaLiZe implication (pre-condition ⇒ post-condition) of the corresponding species:

OCL:

$$\text{context cn :: N(} \text{p}_1 \text{ : type}_1 \text{ .... p}_k \text{ : type}_k\text{)} \tag{9}$$

$$\textbf{pre}: \text{E}_{\text{pre}} \qquad \textbf{post}: \text{E}_{\text{post}}$$

FoCaLiZe:

$$property\ prePostIdent : all\ e : Self;$$
$$all\ p_1 : [[type_1]], \ldots \ldots\ , all\ p_k : [[type_k]\ ], \quad (10)$$
$$[[E_{pre}]] -> [[E_{post}]];$$

- A UML template, a class with general parameters, is transformed into a parameterised species. We transformed each template parameter into a parameter of the corresponding species.

UML:

$$\textbf{public class}\ \mathrm{cn}\ (\mathrm{f\,p_1}\ : \mathrm{type_1}\ , \ldots\ ,\ \mathrm{f\,p_k} : \mathrm{type_k}\ ) = \cdots \textbf{end} \quad (11)$$

FoCaLiZe:

$$\mathrm{species\ sn}\ (\mathrm{f\,p_1\ is\ \ |\ in}\ [[\mathrm{type_1}]]\ , \ldots\ ,\ \mathrm{f\,p_k\ \ is\ |\ in}\ [[\mathrm{type_k}]]) = \cdots \mathrm{end;;} \quad (12)$$

- Each class inheritance is converted into an inheritance between the related species. FoCaLiZe uses clause inherit to specify inheritance between species.

FoCaLiZe, similar to UML, also supports multiple inheritances. Hence, multiple inheritances are also transformed using the same mechanism. Regarding OCL constraints, FoCaLiZe species also inherit properties (transformed from OCL constraints) from their superspecies.

UML:

$$\textbf{public class}\ \mathrm{cn}\ \textbf{inherits}\ \ \mathrm{cn_1}\ , \ldots .\ , \mathrm{cn_k} = \cdots \textbf{end} \quad (13)$$

FoCaLiZe:

$$\mathrm{species\ sn}(\mathrm{p_{sn}}) = \mathrm{inherits\ sn_1}\ (\mathrm{p_{sn1}})\ , \ldots\ , \mathrm{sn_k}(\mathrm{p_{snk}}); \ldots . \mathrm{end;;} \quad (14)$$

- A dependency relationship (supplier-client relationship) between supplier and client classes is transformed into a parameterised species. The client species will be parameterised with the supplier species.

UML:

$$\textbf{public class } \mathrm{cn} \textbf{ depends } (\mathrm{cn}_1, \dots, \mathrm{cn}_k) = \cdots \textbf{ end} \quad (15)$$

FoCaLiZe:

$$\text{species } \mathrm{sn}(\mathrm{p}_{\mathrm{sn1}}, \dots, \mathrm{p}_{\mathrm{snk}}, \ \mathrm{f}\,\mathrm{p}_1 \text{ is } \mathrm{sn}_1(\mathrm{p}_{\mathrm{sn1}}), \dots, \mathrm{f}\,\mathrm{p}_k \text{ is } \mathrm{sn}_k(\mathrm{p}_{\mathrm{snk}})) = \cdots \text{end};; \quad (16)$$

- Parameterized species formalize associations between UML classes in FoCaLiZe. Formal and detailed descriptions of these transformation rules are presented in [9–13].

Following the above transformation rules of the UML class cn into a FoCaLiZe species sn, we update the general transformation as Algorithm 1 states.

Algorithm 1. General transformation of a UML class cn

```
if the class cn already transformed then
        partial_class_transformation ()        ▷ Not the first transformation of the class cn
else
        full_class_transformation()
end if                                         ▷The first transformation of the class cn
```

Source: Authors.

Partial transformation of a class cn, we should update the members and relations of species sn (derived from class cn). In particular, we update the species sn parameters, list of inherited species, list of signatures (derived from attributes and operation), properties (derived from OCL constraints), and proofs (already discharged by a FoCaLiZe developer).

Thus, the partial transformation looks like Algorithm 2 shows.

Algorithm 2. Partial transformation of a UML class cn

```
                                        ▷ This transformation updates the species sn
update_species_parameters()             ▷ derived from the class cn
update_species_inherited_Species_List()
update_species_signatures()
update_species_properties()
update_species_proofs()
```

Source: Authors.

When applying the partial transformation of one class cn, the transformation of any other class cn′ that depends on the class cn must be updated. Therefore, we also call the partial transformation of cn′. We distinguish several cases in which a class cn′ depends on the class cn:

$cn \in H_{cn'}$, the class cn is a parent of the class cn′, inheritance.

$cn \in P_{cn'}$, the class cn is a parameter of the class cn′.

$cn \in D_{cn'}$, the class cn′ has a dependency relationship with the class cn′.

$cn \in T_{cn'}$, the class cn′ is a bound class derived from the class cn by substitution of its formal parameters with actual ones.

In all the cases mentioned above, the species sn′ derived from the class cn′ must be updated.

In the following, we describe algorithms for the partial transformation of class attributes and operations.

Algorithm 3. Partial transformation of attributes

```
foreach attribute_i ∈ A_cn do
    signature_i ← [[ attribute_i ]]
    if signature_i ∈ Sig_sn then
        if changed (signature_i) then
            replace (signature_i)
            update_Prop_sn(signature_i)
            erase_species_proofs(signature_i)
        end if
    else
        insert(signature_i)
    end if
end foreach
```

▷ we get the transformation of attribute$_i$ in signature$_i$

▷ the signature$_i$ already existed in Sig$_{sn}$

▷signature is different from the ▷old occurrence of signature

▷ replace the old signature$_i$ with the new one

▷ update all sn properties referring to signature$_i$

▷ erase all species proofs using signature$_i$

▷insert the new signature signature$_i$ into Sig$_{sn}$

Source: Authors.

In Algorithm 3, the statement update_Propsn ($signature_i$) aims to modify all properties included in Propsn using the signature $signature_i$. The functional type changes because the signature $signature_i$ keeps its identifier. For the same reason, all the related proofs should be redone by a FoCaLiZe developer, and thus, the old ones must be erased.

Similarly, we define Algorithm 4 to update operation transformation.

Algorithm 4. Partial transformation of operations

```
foreach op_i ∈ O_cn do
    signature_i ← [[op_i]]                          ▷ we get the transformation of op_i in signature_i

    if signature_i ∈ Sig_sn then                    ▷ the signature_i already existed in Sigsn
        if changed(signature_i) then                ▷ New signature_i is different from the old signature_i
            replace(signature_i)                    ▷ replace the old occurrence signature_i with the new one
            Update_Prop_sn(signature_i)             ▷ update all sn properties referring to signature_i
            Erase_species_proofs(signature_i)       ▷ erase all species proofs using signature_i
        end if
    else
        insert(signature_i)                         ▷insert the new signature signature_i into Sigsn
    end if
end foreach
```

Source: Authors.

Both partial transformation of attributes and operations (Algorithm 3 and Algorithm 4) are parts of the subroutine update_species_signatures().

3.3 DEVELOPMENT STAGES OF THE MILITARY OBJECTS SYSTEM

We present now the development stages of the military objects system illustrated in Fig. 2. Through this example, we highlight how the partial transformation significantly affects the development process in two cases: Firstly, when a UML/OCL error occurs by the modeller, and secondly, when the client introduces a new increment (requirements).

Case1: When a UML/OCL error occurred by the modeller.

Let us examine the incorrect class Object_2D presented in Fig. 4. The UML modeller mistakenly specifies the method distance that calculates the distance between two military objects. He provides two parameters of class Object_2D, as

follows: + distance (obj1: Object_2D, obj2: Object_2D): float. However, the method distance is a binary method that only requires one object of the class Object_2D as a parameter. The second object is the object that will invoke the method: (the correct method call is Obj1.distance (Obj2))

On the other hand, the UML modeller provides a correct OCL invariant when specifying the method diastance:

inv distance_specification: allInstances()-> forAll(obj1,obj2 | obj1.equal(obj2) implies obj1.distance(obj2) = 0.00)

In this OCL invariant, the operation call is correct (obj1.distance(obj2)), but contra dictory with the method distance signature.

The first time we launch the transformation (complete transformation) of the incorrect class Object_2D, we get Code 1. The texts between "(*" and "*)" show comments.

Code 1. The first transformation of the incorrect class Object_2D

```
(*This statement opens the standard FoCaLiZe library *)
open "basics";;

(* The species Object_2D transforms the class Object_2D *)
species Object_2D =


(*Transformation of the class Object_2D attributes *)
signature get_abscissa : Self -> float;
signature get_ordinate : Self -> float;

(*Transformation of the class Object_2D methods *)
signature new_object : float -> float -> Self ;
signature distance : Self -> Self -> Self -> float;
signature equal : Self -> Self -> bool;
signature different : Self -> Self -> bool;
signature print_object : Self -> string ;

(*Transformation of the class Object_2D constraints *)
property equal_reflexive : all obj:Self, equal(obj, obj);
property equal_symetric: all obj1 obj2: Self, equal(obj1, obj2)->
                                          equal(obj2, obj1);
```

```
property equal_transitive : all obj1 obj2 obj3: Self,
                          ( equal(obj1, obj2) ->
                            equal(obj2, obj3)) ->
                           equal(obj1, obj3);
property equal_is_not_different : all obj1 obj2:Self,
                            equal(obj1, obj2) ->
                            ~(different(obj1, obj2)) ;
property distance_specification : all obj1 obj2:Self,
                        equal(obj1, obj2) ->
                              (distance(obj1, obj2) = 0.00) ;
end;;
```

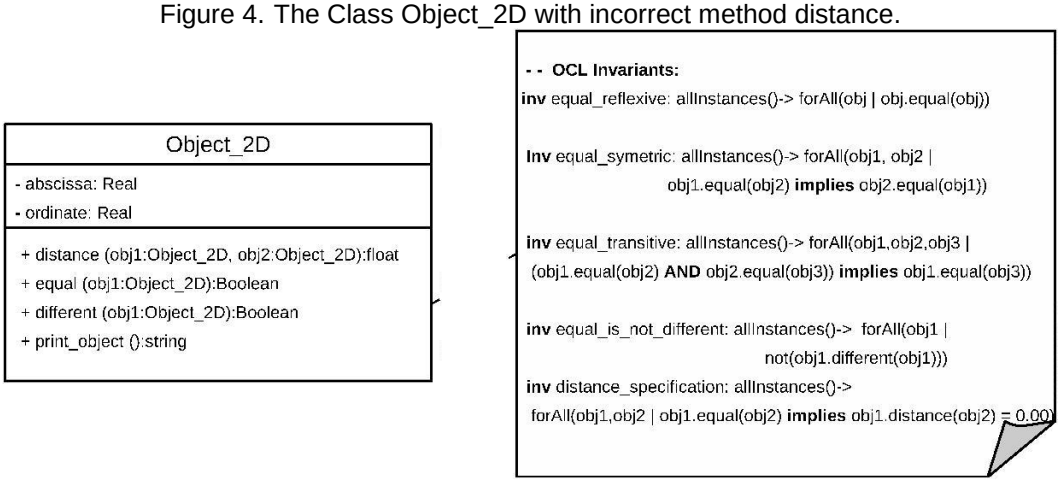
Source: Authors.

As we can notice in Code 1, the transformations of the method diastance is also incorrect. It follows the UML/OCL mistake. However, the transformation of the OCL specification (the invariant distance_specification) is correct, but it is contradictory with the method diastance signature.

When the FoCaLiZe developer compiles the derived code, he gets the following error message:

~ /Documents/CODES/Military/Point_Test$ focalizec militaryObj.fcl
File "militaryObj.fcl", line 125, characters 57-84:

Figure 4. The Class Object_2D with incorrect method distance.



Source: Authors.

Error: Types basics#float and Self -> basics#float are not compatible.

When correcting the UML/OCL mistake (distance with one object parameter: + distance (obj1:Object_2D):float) and relaunch the transformation again; thanks to partial transformation, only the method distance will be updated in the generated species.

Imagine now that a FoCaLiZe developer interacts and completes the species Object_2D derived from the class Object_2D, see Code 2. Hence, he provides definitions for the generated methods and discharge proofs for the generated properties (from OCL invariants). In this case, the loss will be great. The FoCaLiZe developer should redo all the previously developed code. Otherwise, he should manage his codes.

Code 2. The refined species Object_2D

```
(*The species Objet_2D completed by a FoCaLiZe developer *)
species Objet_2D =

(* The representation type of the species Objet_2D *)
representation = float * float;

(*Methods definition of the species Objet_2D *)
let get_abscissa(obj) = fst(obj);
let get_ordinate(obj) = snd(obj);
let new_object(abs: float, ord: float): Self = (abs , ord);
let move_object(obj, d_abscissa, d_ordinate): Self =
    ( get_abscissa(obj) + d_abscissa,
      get_ordinate(obj) + d_ordinate );

(* distance follows the incorrect specification. It is incorrect regarding to
its OCL invariant *)

let distance (obj1:Self, obj1:Self, obj2: Self):float =
  # sqrt(  ( get_abscissa(obj1) - get_abscissa(obj2) )*
         ( get_abscissa(obj1) - get_abscissa(obj2) ) +
         ( get_ordinate(obj1) - get_ordinate(obj2) )  *
         ( get_ordinate(obj1) - get_ordinate(obj2) ));



let equal (obj1:Self, obj2: Self):bool =
    (get_abscissa(obj1) = get_abscissa(obj2)) &&
    (get_ordinate(obj1) = get_ordinate(obj2));

let different (obj1:Self, obj2: Self):bool =
                          ~~(equal(obj1, obj2));

let print_object (obj:Self):string = (" Abscissa = " ^
           string_of_float(get_abscissa(obj)) ^
           " Ordinate = " ^ string_of_float(get_ordinate(obj)));

(*Profs of the species Objet_2D properties*)
proof of equal_reflexive = by definition of * equal;
proof of equal_symetric = by definition of equal;
proof of equal_transitive =
                by definition of equal property equal_symetric;
proof of equal_is_not_different = by definition of different;
proof of distance_specification = by definition of distance equal
   property equal_symetric equal_reflexive equal_transitive ;
end;;
```

Source: Authors.

The FoCaLiZe developer provides computational bodies for methods and proof hints (indications) for the derived properties in this code. Unfortunately, when compiling the code, the proof of the property distance_specification will lack because of the incompatibilities between the method distance and its OCL invariant. The error message shown by FoCaLiZe leads developers to correct the UML/OCL model and launch a partial transformation. Thus, thanks to partial transformation, only the specification (signature) and the definition (let distance ...) of the method distance will be updated. Also, the proof of the property distance_specification will be erased and has to be redone.

Case2: When the client introduces new increments

As Fig. 1 states, the initial military objects system considers only military objects on the surface (Object_2D). After the delivery of this initial system, the client would like to consider objects in the space (Object_3D, see Fig. 2). So, let us complete the development process of the first increment. Then, we highlight how the new increment will be achieved using partial transformation features.

The refined species Object_2D derived from the class Object_2D has already been presented in code 2. Code 3 shows the abstract species Controller derived from the class Controller. The latter has a unidirectional association with the class Object_2d. It expresses the group of objects controlled by the controller. Therefore, the species Controller is parameterised by the species Object_2D to define its own methods and properties (see code 3).

Code 3. The FoCaLiZe Abstract specification generated from the UML/OCL model military objects system

```
(* The species Controller generated from the class Controller *)
species Controller (Obj is Object_2D) =
    (* This is the controller object*)
signature controller_obj: Self -> Obj;
(*This is the group of controlled objects *)
signature group: Self -> list(Obj);
(* This is the max distance between the controller and objects *)
signature max_distance: Self -> float;
(* constructor method )
signature create_controller: Obj -> float -> Self;
signature add_object_to_controller: Self -> Obj -> Self;
signature remove_object_from_controller: Obj -> Self -> Self;
signature belongs: Obj -> list(Obj) -> bool;
signature move_object: Self -> Obj -> float -> float -> Obj;

property add_object_specification : all c:Self, all obj:Obj,
        belongs(obj, group(add_object_to_controller(c, obj)) );
property belongs_specification : all c:Self, all obj:Obj,
 belongs(obj, group(c)) ->
 Obj!distance(obj, controller_obj(c)) <= max_distance(c);
property move_specification :
 all obj:Obj, all v1 v2:float, all c:Self,
 belongs(obj, group(c)) ->
 Obj!distance(move_object(c, obj, v1, v2), controller_obj(c))
<= max_distance(c);
end;;
```

Source: Authors.

In the next step, a FoCaLiZe developer completes the previously generated code, and he implements all methods of the species controller and discharge its proofs (for brevity, only the proof of the property move_specification is detailed here). The refined species controller is presented in code 4.

Code 4. Refinement of the species Controller

```
(*definition of the species Controller methods and properties *)
species Controller(Obj is Object_2D) =
 representation = (Obj    list(Obj)) float;
let controller_obj(c:Self):Obj = fst(fst(c));
let group(c:Self):list(Obj) = snd(fst(c));
let max_distance(c:Self):float = snd(c);
    let equal (c1:Self, c2:Self): bool = controller_obj(c1) = controller_obj(c2) &&
                    (group(c1) = group(c2)) &&
        max_distance(c1) = max_distance(c2) ;

let emptyGroup:list(Obj)=[];
let create_controller (obj:Obj, max_distance:float):Self =
    ((obj, emptyGroup), max_distance);
let add_object_to_controller(c: Self, obj:Obj):Self =
    if (Obj!distance(obj, controller_obj(c)) <= max_distance(c))
    then ((obj, group(c)), max_distance(c))
    else c;
let rec delete_object (obj:Obj, g:list(Obj)):list(Obj) =
      match g with
      | [] -> []
      | q::r ->if (Obj!equal(q, obj)) then r
                else q::(delete_object(obj, r))
     termination proof = structural g ;

let remove_object_from_controller (obj:Obj, c: Self):Self =
( (controller_obj(c), delete_object(obj,group(c)) ),
                                          max_distance(c));

let rec belongs (obj:Obj, l:list(Obj)):bool = match l with
|[] -> false
|q::r ->if (Obj!equal(q, obj)) then true else belongs(obj, r)

termination proof = structural l ;

let move_object(obj:Obj, d_abscissa:float,
                                  d_ordinate:float, c:Self):Obj =
let obj_1 = Obj!new_object( Obj!get_abscissa(obj) + d_abscissa,
                    Obj!get_ordinate(obj) + d_ordinate ) in
if Obj!distance(obj_1, controller_obj(c)) <= max_distance(c)
then obj_1  else obj;

theorem move_specification: all obj:Obj, all v1 v2:float,
                              all c:Self,
        belongs(obj, group(c)) -> Obj!distance(move_object
       (c, obj, v1, v2), controller_obj(c)) <= max_distance(c)
proof = by definition of move_object property
belongs_specification;
end;;
```

Source: Authors.

The system may be delivered to final users after the species Object_2D and Controller are completed. Entities from the final species could be created and used by the final user, as shown in code 5 below.

Code 5. Using the initial military objects system

```
(*create a new collection *)
collection  Object2D_Collection =
                        implement Factual_Objet_2D; end;;
(* create a new object 2D *)
let obj2D_1 = Object2D_Collection!new_object(2.0, 5.0);;
(* create a new second object 2D*)
let obj2D_2 = Object2D_Collection!new_object(5.0, 9.0);;
(* create a new third object 2D*)
let obj2D_3 = Object2D_Collection!new_object(0.0, 0.0);;

(*create a new collection from controller*)
collection  Controller_Collection = implement
            Controller(Object2D_Collection); end;;
(*create a new new controller controller1 *)
let controller1 =
    Controller_Collection!create_controller(obj2D_3, 1.0);;
(*Add objects to controller1)
let controller2 = Controller_Collection!add_object_to_controller
                            (obj2D_1, controller1);;
let controller3 =
Controller_Collection!add_object_to_controller
(obj2D_2, controller2);;
```

Source: Authors.

Now, after the FoCaLiZe code is tested and delivered to the client, a new increment is required. It consists of considering military objects in the space (Object_3D), in addition to objects on the surface (Object_3D) At UML/OCL level, the class diagram of the model is amended as in Fig 2. Due to the proposed partial transformation features, we only need to relaunch the transformation of the new model. Then, the initial FoCaLiZe code will be automatically amended without losing any of the FoCaLiZe developer efforts. At FoCaLiZe level, a new increment (the species Object_3D) is kindly inserted as shows code 6.

Code 6. The refined new increment Object_3D

```
species Object_3D = inherit Object_2D;
  representation = (float * float) float;
let get_abscissa(obj : Self) : float = fst(fst(obj));
let get_ordinate(obj : Self) : float = snd(fst(obj));
let get_altitude(obj : Self) : float = snd(obj);
let new_object3D(abs: float, ord : float, alt : float) : Self = ((abs , ord), alt);
let move_object3D(obj, d_abscissa, d_ordinate, d_altitude) : Self =
        ( ( get_abscissa(obj) + d_abscissa ,
            get_ordinate(obj) + d_ordinate ),
            get_altitude(obj) + d_altitude
let distance (obj1 : Self, obj2 : Self) : float =
# sqrt(  ( get_abscissa(obj1) − get_abscissa(obj2) )*
         ( get_abscissa(obj1) − get_abscissa(obj2) ) +
             ( get_ordinate(obj1) − get_ordinate(obj2) )*
             ( get_ordinate(obj1) − get_ordinate(obj2) ) +
             ( get_altitude(obj1) − get_altitude(obj2 ))*
             ( get_altitude(obj1) − get_altitude(obj2)  ) );
let equal (obj1 : Self, obj2 : Self) : bool =
   (get_abscissa(obj1) = get_abscissa(obj2)) &&
   (get_ordinate(obj1) = get_ordinate(obj2)) &&
   (get_altitude(obj1) = get_altitude(obj2)) ;
let print_object (obj : Self) : string =
     (" Abscissa = " ^ string_of_float(get_abscissa(obj)) ^
      " Ordinate = " ^ string_of_float(get_ordinate(obj)) ^
      “Altitude = " ^ string_of_float(get_altitude(obj)) );
let different (obj1 : Self, obj2 : Self) : bool = ~ (equal(obj1, obj2));

proof of  equal_reflexive = by definition of equal;
proof of  equal_symetric = by definition of equal;
proof of equal_transitive = by definition of equal
                            property equal_symetric;

proof of equal_is_not_different = by definition of different;
end; ;
```

Source: Authors.

The species Objects_3D inherits the species Object_2D. It inherits all its methods, theorems, and proofs. However, all the inherited proofs are cleared (erased) since most inherited methods are redefined. The partial transformation forces the proof of theorems to be redone when methods are redefined. All the previously developed code is maintained and still usable without additional intervention. The newly generated and refined species (Object_3D) is also used together with the species Controller. Otherwise, the new Controller may control a group of Object_3D in the same way as a group of objects of the species Object_2D (see code 7)

Code 7. Using the newly developed increment

```
(* Here, we use the 3D objects to create new
  controller and add 3D objects to it *)

collection Object3D_Collection = implement Object_3D; end;;
let obj3D_1 = Object3D_Collection!new_object3D(2.0, 5.0, 3.5);;
let obj3D_2 = Object3D_Collection!new_object3D(5.0, 9.0, 5.5);;
let obj3D_3 = Object3D_Collection!new_object3D(0.0, 0.0, 0.0);;
 collection Controller2_Collection = implement
          Factual_Controller(Object3D_Collection); end;;
let controller4 = Controller2_Collection! create_controller
                  (obj3D_3, 4.0);;
let controller5 = Controller2_Collection! add_object_to_controller
                  (obj3D_1, controller4);;
 let controller6 = Controller2_Collection! add_object_to_controller
                  (obj3D_2, controller5);;
```

Source: Authors.

As we can conclude here, without the partial transformation contribution, the maintenance phase will be difficult, and the FoCaLiZe developer should manage the development to avoid losing the manually updated code. Using the partial transformation, we need only to relaunch the transformation, and all the generated code (at the preceding transformations) will be maintained and updated correctly.

Fig. 3 presents the general lifecycle process. More details about the transformation tool (UML2FOC) that implements the transformation and partial transformation rules from UML/OCL to FoCaLiZe are available at https://github.com/MessaoudAbbas/UML2FOC.

The complete FoCaLiZe development of the military objects example is available at https://github.com/MessaoudAbbas/Codes_Military_Objects/blob/main/Military.fcl.

4 RELATED WORK

The FoCaLiZe environment has been developed to meet the development needs of critical systems.

[11] Show how to develop critical systems using the FoCaL environment. They illustrate how FoCaL could help to fulfil requirements using a complete example.

[36] Describe how a FoCaLiZe developer could formally specify secure systems through an incremental refinement of the design and implementation.

They propose a generic framework to define and certify security policies using modular implementation.

[73] Illustrates how software engineers should integrate a tool for graphical modeling, such as UML, where non-specialized partners may be involved.

In our work, we combine UML/OCL with the FoCaLiZe framework in a complete development lifecycle for critical systems, which enhances the previous work.

Several other formal methods have been used in the development of critical systems. We could refer to the method B [8] and Event-B [7], Alloy [51], Maude [26], Petri Nets [48], and the language Z [63], among these formal methods (but not limited to).

[69] Proposed a rigorous approach for developing critical systems using Event-B.

[34] Presented a diagrammatic Event-B formal modeling of a hybrid, 'fixed virtual block' approach, to train movement control for the emerging European Rail Traffic Management System.

[30,55,74] also show a combination of Event-B and UML/OCL. They focused on transformation rules and verification approaches of constraints of the UML model using Event-B and its proof techniques. However, they did not show a general lifecycle approach with all development phases.

[67] Propose an approach that covers most phases of software development, from modelling and verification to code generation. The authors aim to provide agility during the designing of systems. They transform a UML class diagram and its state diagram that describes a system's static and dynamic aspects into B method specifications. Then, they generate a C code after refining and verifying constraints.

When FoCaLiZe is compared with Event-B, FoCaLiZe could be used naturally to specify UML class diagrams. Event-B does not have a lifting mechanism to specify the characteristics of a set of instances. FoCaLiZe supports inheritance, dependencies, and late binding between species, whereas Event-B strongly supports refinement, which helps specify inheritance. However, Event-B developers have to encode inheritance via subsets of instances explicitly. Object-oriented features are required for an incremental design and architecture.

Hence, transforming UML/OCL models into Event-B leads to non-similar models, and thus, we lose properties to be verified. On the other hand, FoCaLiZe requires developer assistance during the proof, which is not always required during the proof of constraints and refinements using Event-B.

Petri Nets (PNs) are a graphical formalism used to specify and verify the comportment of concurrent and distributed systems [48]. PNs were used for several applications. [24] Translated UML sequence diagram into Time PNs for verification purposes. [62] trans- formed UML state chart diagrams into Time Coloured PNs. [17] run and reconfigured Python applications under the supervision and control of PNs with Graph Rewriting Rules (GRRs) using a model at runtime technique. GRRs were used to change the structure of the PNs and, thus, the behaviour of the Python application. [16] Have used PNs to determine instances of a business process that support change. GRRs partially transform Python applications and business processes when new requirements appear. These approaches are iterative. However, the verification considers only the high-level specification using model checking. [18] Have proposed a dynamic Colored PN Model to obtain routes in the presence of obstacles detected using a computer vision technique.

Alloy formal language has a syntax that is compatible with graphical object models. It is used to describe structural properties using a set-based formula syntax. [53] Proposed an approach that transforms UML activity diagrams into Alloy, and then the Alloy-Analyzer is used to check model properties. [52] Proposed an interactive transformation approach between the UML class diagram and Alloy.

The Maude system [25] is a language that implements rewriting logic. It is used in various approaches, such as in [35]. The latter proposes an approach for formalizing the Interaction Overview Diagram from UML 2.0 into the Maude rewriting logic language.

This formalization allows the specification and formal analysis of the dynamic behaviour of the built system.

Finally, the formal specification language Z was also recently used [59] to formalize and verify UML models. The behaviours of systems are specified by use

case diagrams and activity diagrams and then transformed into Z formal specification. The derived specifications are checked by the Z/EVES prover[5].

In summary, thanks to FoCaLiZe features, the lifecycle approach presented in this article is the most complete. It considers most development stages: requirements specification, architecture design, implementation, testing, maintenance, and documentation. The proposed partial transformation from UML/OCL to FoCaLiZe assists in the maintenance phase and incremental development.

5 CONCLUSION AND PERSPECTIVES

In this paper, we have proposed a development lifecycle for critical systems. We combined UML/OCL and FoCaLiZe formal methods. We have covered all stages of development, from specification and design to implementation, testing, validation, and proving. At the maintenance phase, we provided an interactive transformation between UML/OCL and FoCaLiZe, using a partial transformation from UML/OCL to FoCaLiZe. Design and implementation are the main activities in the proposed critical software process while testing and verification are incorporated along with the development phases. Incremental development is also enabled through this development process where new requirements are easy to fit into each new increment. Each new stable increment can be delivered to customers, and then we take their feedback into consideration in the next increment.

Moreover, strict boundaries and traceability between UML/OCL modeling and FoCaLiZe environment are ensured. However, no other formalism is used in the proposed process except UML/OCL and FoCaLiZe.

The findings of this research offer significant contributions both to society and the academic community. Societally, our approach proposes a robust framework for the development of critical systems, which could enhance the security, reliability, and sustainability of various infrastructures and applications, such as military control systems, healthcare systems, and transportation systems. Moreover, in an academic context, this research opens new avenues for the study and practice of formal methods in critical software development, providing researchers and students with a methodological framework and

concrete case studies for future work. Lastly, it is important to note some limitations of this research. Our approach primarily relies on the integration of UML/OCL and FoCaLiZe, which may limit its generalizability to other methods or development environments. Additionally, the complexity of partial transformation between UML/OCL and FoCaLiZe may require additional efforts in terms of development and maintenance. Therefore, for future work, we recommend further exploration of tools and formalisms, as well as investigating opportunities for automation and optimization of the transformation processes. Furthermore, empirical studies to evaluate the effectiveness and efficiency of our approach in real-world contexts of critical system development could provide valuable insights for its continuous improvement.

As an extension to this work, we would like to consider more UML/OCL tools as a starting point for developing critical systems. This may be achieved by establishing an abstract syntax (EBNF syntax) for UML/OCL that should play an intermediate role (like Java byte code) between UML/OCL and FoCaLiZe. Although the XMI is an OMG standard, we observed significant differences between the XMI implementations within UML tools.

Consequently, we initially transformed the UML/OCL model into the proposed EBNF syntax before proceeding with the transformation into a FoCaLiZe in order to complete the remaining phases. UML use cases and activity diagrams will be considered within the intended extension.

REFERENCES

[1] ABBAS, M.; BEN-YELLES, C. B.; RIOBOO, R. Formalizing UML/OCL multiple inheritance with FoCaLiZe. *In*: **Proceedings of the 2018 International Conference on Smart Communications in Network Technologies (SaCoNeT)**. IEEE, 2018. p. 261–266.

[2] ABBAS, M.; BEN-YELLES, C. B.; RIOBOO, R. Formalizing UML/OCL structural features with FoCaLiZe. **Soft Computing**, v. 24, p. 4149–4164, 2020.

[3] ABBAS, M.; BEN-YELLES, C. B.; RIOBOO, R. Modeling UML template classes with FoCaLiZe. *In*: **Proceedings of the International Conference on Integrated Formal Methods**. Springer, 2014.

[4] ABBAS, M.; BEN-YELLES, C. B.; RIOBOO, R. Modelling UML state machines with FoCaLiZe. International Journal of Information and Communication Technology, v. 13, p. 34–54, 2018.

[5] ABBAS, M.; HALOUA, F.; BOUCHERIT, A. Development of Critical Systems with UML/OCL and FoCaLiZe. *In*: **Proceedings of the International Conference on Model and Data Engineering**. Springer, 2021. p. 16–30.

[6] ABBAS, M.; RIOBOO, R.; BEN-YELLES, C. B.; SNOOK, C. F. Formal modeling and verification of UML Activity Diagrams (UAD) with FoCaLiZe. **Journal of Systems Architecture**, v. 114, p. 101911, 2021.

[7] ABRIAL, J. R. **Modeling in Event-B**: system and software engineering. Cambridge University Press, 2010.

[8] ABRIAL, J. R. **The B-Book**: Assigning Programs to Meanings. Cambridge University Press, 2005.

[9] ANASTASAKIS, K.; BORDBAR, B.; GEORG, G.; RAY, I. UML2Alloy: A Challenging Model Transformation. *In*: **Model Driven Engineering Languages and Systems**. Springer, 2007. p. 436–450.

[10] ANWAR, A. A review of rup (rational unified process). **International Journal of Software Engineering (IJSE)**, v. 5, p.12–19, 2014.

[11] AYRAULT, P.; HARDIN, T.; PESSAUX, F. Development Life-Cycle of Critical Software under FoCal. **Electronic Notes in Theoretical Computer Science**, v. 243, p. 15–31, 2009.

[12] BENEDICENTI, L.; MESSINA, A.; SILLITTI, A. iAgile: mission critical military software development. *In*: **Proceedings of the 2017 International Conference on High Performance Computing & Simulation (HPCS)**. IEEE, 2017. p. 545–552.

[13] BENMAN, E.; AIMEN, D. Toward the Development of a Unified Process and Methodology Guide for Environmental Justice Analysis in Planning and Programming. **Transportation Research Record**, p. 03611981211028868, 2021.

[14] BONICHON, R.; DELAHAYE, D.; DOLIGEZ, D. Zenon: An extensible automated theorem prover producing checkable proofs. *In*: **Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning**. Springer, 2007. p. 151–165.

[15] BONICHON, R.; DELAHAYE, D.; DOLIGEZ, D. Zenon: An extensible automated theorem prover producing checkable proofs. *In*: **Proceedings of the LPAR**. Springer, 2007. v. 4790, p. 151–165.

[16] BOUHAMED, M.M.; CHAOUI, A.; NOUARA, R.; DÍAZ, G.; DEMBRI, A. Reducing the number of migrated instances during business process change: A graph rewriting approach. Journal of King Saud University-Computer and Information Sciences, v. 34, p. 7720–7734, 2022.

[17] BOUHAMED, M. M.; DÍAZ, G.; CHAOUI, A.; KAMEL, O.; NOUARA, R. Models@ Runtime: The Development and Re-Configuration Management of Python Applications Using Formal Methods. **Applied Sciences**, v. 11, p. 9743, 2021.

[18] BOUZENADA, A.; BOUHAMED, M. M.; KAMEL, O.; MACIÀ, H.; DÍAZ, G.; CHAOUI, A. An Example of a Dynamic CPN Model to Obtain Routes in the Presence of Obstacles Detected Using Machine Learning Techniques. *In*: **Proceedings of the International Symposium on Modelling and Implementation of Complex Systems**. Springer, 2023. p. 220–233.

[19] BROWN, S. Overview of IEC 61508. Design of electrical/ electronic/ programmable electronic safety-related systems. **Computing & Control Engineering Journal**, v. 11, p. 6–12, 2000.

[20] BROY, M.; CRANE, M.; DINGEL, J.; HARTMAN, A.; RUMPE, B.; SELIC, B. Formal Semantics for UML. **Models in Software Engineering**, 2007. p. 318–323.

[21] BRUCKER, A. D.; WOLFF, B. HOL-OCL: A Formal Proof Environment for UML/OCL. *In*: **Fundamental Approaches to Software Engineering**. Springer, 2008. p. 97–100.

[22] CABOT, J.; PAU, R.; RAVENTÓS, R. From UML/OCL to SBVR specifications: A challenging transformation. **Information systems**, v. 35, p. 417–440, 2010.

[23] CARLIER, M.; DUBOIS, C. Functional testing in the focal environment. *In*: **Proceedings of the International Conference on Tests and Proofs**. Springer, 2008. p. 84–98.

[24] CHABBAT, N.; SAIDOUNI, D.E.; BOUKHARROU, R.; GHANEMI, S. Formal Verification of UML MARTE Specifications Based on a True Concurrency Real Time Model. **Computing and Informatics**, v. 39, p. 1022–1060, 2020.

[25] CLAVEL, M.; DURAN, F.; EKER, S.; LINCOLN, P.; MARTI-OLIET, N.; MESEGUER, J.; QUESADA, J. F. The Maude system. *In*: **Proceedings of the International Conference on Rewriting Techniques and Applications**. Springer, 1999. p. 240–243.

[26] CLAVEL, M.; DURÁN, F.; EKER, S.; LINCOLN, P.; MARTÍ-OLIET, N.; MESEGUER, J.; TALCOTT, C. **All about Maude a High Performance Logical Framework**: How to Specify, Program and Verify Systems in Rewriting Logic. Springer-Verlag, 2007.

[27] COQ. The Coq Proof Assistant, Tutorial and Reference Manual, Version 8.4. INRIA – LIP – LRI – LIX – PPS 2012. Distribution available at: http://coq.inria.fr/.

[28] CUCCURU, A.; RADERMACHER, A.; GÉRARD, S.; TERRIER, F. Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers. *In*: **Model Driven Engineering Languages and Systems**. Springer, 2009. p. 644–649.

[29] CUNHA, A.; GARIS, A.; RIESCO, D. Translating between Alloy Specifications and UML Class Diagrams Annotated with OCL Constraints. **Software & Systems Modeling**, p. 1–21, 2013.

[30] DE SOUSA, T. C.; KELVIN, L.; NETO, C. D.; DE CARVALHO, C. G. N. A Formal Semantics for Use Case Diagram Via Event-B. **J. Softw.**, v. 12, p. 189–200, 2017.

[31] DE-LA HOZ-HERNÁNDEZ, J. D.; TRONCOSO-PALACIO, A.; DE-LA HOZ-FRANCO, E. Implementation of the Eclipse Process Framework Composer Tool for the Documentation of Quality Management Systems: A Case Applied in Healthcare Services. *In*: **Proceedings of the International Conference on Swarm Intelligence**. Springer, 2019. p. 200–210.

[32] DELAHAYE, D.; ÉTIENNE, J.; DONZEAU-GOUGE, V. Producing UML Models from Focal Specifications: An Application to Airport Security Regulations. *In*: **Proceedings of the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering**, 2008. p. 121–124.

[33] DELAHAYE, D.; ÉTIENNE, J.; DONZEAU-GOUGE, V. Producing UML Models from Focal Specifications: An Application to Airport Security Regulations. *In*: **Proceedings of the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering**, 2008. p. 121–124.

[34] DGHAYM, D.; POPPLETON, M.; SNOOK, C. Diagram-led formal modelling using iUML-B for hybrid ERTMS level 3. *In*: **Proceedings of the International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z**. Springer, 2018. p. 338–352.

[35] DJAOUI, C.; KERKOUCHE, E.; CHAOUI, A.; KHALFAOUI, K. A Graph Transformation Approach to Generate Analysable Maude Specifications from UML Interaction Overview Diagrams. *In*: **Proceedings of the 2018 IEEE International Conference on Information Reuse and Integration (IRI)**. IEEE, 2018. p. 511–517.

[36] DOLIGEZ, D.; JAUME, M.; RIOBOO, R. Development of secured systems by mixing programs, specifications and proofs in an object-oriented programming environment: a case study within the FoCaLiZe environment. *In*: **Proceedings of**

the Proceedings of the 7th Workshop on Programming Languages and Analysis for Security, 2012. p. 1–12

[37] DUBOIS, C.; HARDIN, T.; DONZEAU-GOUGE, V. Building Certified Components within FOCAL. **Trends in Functional Programming**, v. 5, p. 33–48, 2006.

[38] DURÁN, F.; GOGOLLA, M.; ROLDÁN, M. Tracing Properties of UML and OCL Models with Maude.arXiv preprint arXiv: 1107.0068 2011.

[39] EDEKI, C. Agile unified process. **International Journal of Computer Science**, v. 1, p. 13–17, 2013.

[40] ESCHBACH, R. Industrial application of Event-B to a wayside train monitoring system: Formal conceptual data analysis. *In*: **Proceedings of the International Symposium on Formal Methods**. Springer, 2019. p. 738–745.

[41] FACON, P.; LALEAU, R. Des spécifications informelles aux spécifications formelles : compilation ou interprétation? **Actes du 13ème congrès INFORSID**, 1995.

[42] FECHTER, S. **Sémantique des Traits Orientés Objet de Focal**. Thesis (Doctorate) – Paris 6, 2005.

[43] FRANÇOIS, P. **Another Tutorial for FoCaLize**: Playing with Proofs. 2013.

[44] GU, J.; CHEN, M.; X., Z. Formal Language B and UML/OCL Comparison. **Computer Knowledge and Technology**, v. 34, 2009.

[45] HADAD, A. S. A.; MA, C.; AHMED, A. A. O. Formal Verification of AADL Models by Event-B. **IEEE Access**, v. 8, p. 72814–72834, 2020.

[46] HARDIN, T.; FRANCOIS, P.; PIERRE, W.; DAMIEN, D. **FoCaLiZe**: Tutorial and Reference Manual, version 0.9.2. CNAM/INRIA/LIP6 2018. available at: http://focalize.inria.fr.

[47] HAZEM, L.; LEVY, N.; MARCANO-KAMENOFF, R. UML2B: Un Outil pour la Génération de Modèle Formels. *In*: **Proceedings of the AFADL**, 2004.

[48] HE, X. PZ nets—a formal method integrating Petri nets with Z. **Information and Software Technology**, v. 43, p. 1–18, 2001.

[49] INFRASTRUCTURE, P. K.; PROfiLE, T. P. **Common criteria for information technology security evaluation**. National Security Agency 2002.

[50] JACKSON, D. **Software Abstractions**: Logic, Language and Anlysis; MIT press, 2012.

[51] JACKSON, D. **Software Abstractions**: logic, language, and analysis; MIT press. Cambridge: Great Britain, 2012.

[52] KAUTZ, O.; MAOZ, S.; RINGERT, J. O.; RUMPE, B. CD2Alloy: a translation of class diagrams to Alloy. Techn. Rep. AIB-2017-06, RWTH Aachen University (July 2017) 2017.

[53] KHERBOUCHE, M.; MOLNÁR, B. Formal Model Checking and Transformations of Models Represented in UML with Alloy. *In*: **Proceedings of the International Workshop on Modelling to Program**. Springer, 2020. p. 127–136.

[54] KIENZLE, J.; AL ABED, W.; FLEUREY, F.; JÉZÉQUEL, J. M.; KLEIN, J. Aspect-oriented Design with Reusable Aspect Models. *In*: **Transactions on aspect-oriented software development VII**. Springer, 2010. p. 272–320.

[55] KRAIBI, K.; AYED, R.B.; COLLART-DUTILLEUL, S.; BON, P.; PETIT, D. Analysis and Formal Modeling of Systems Behavior Using UML/Event-B. **Journal of Communications**, v. 14, p. 980–986, 2019.

[56] LEDANG, H.; SOUQUIÈRES, J.; CHARLES, S. *et al*. ArgoUML+ B: Un Outil de Transformation Systématique de Spécifications UML en B. *In*: **Proceedings of the Approches Formelles dans l'Assistance au Développement de Logiciels**, 2003.

[57] MITHUN, P.; SAMUEL, A.A.; RANJAN, P.; JAYALAL, N.; GOPALAKRISHNAN, T.; VALSA, B. A Framework for Test Coverage of Safety and Mission Critical Software. *In*: **Proceedings of the Second International Conference on Networks and Advances in Computational Technologies**. Springer, 2021. p. 37–47.

[58] MOKHATI, F.; SAHRAOUI, B.; BOUZAHER, S.; KIMOUR, M. T. **A Tool for Specifying and Validating Agents' Interaction Protocols**: From Agent UML to Maude. Object Technology 2010, 9.

[59] MUHAMAD, Z. H.; ABDULMONIM, D. A.; ALATHARI, B. An integration of uml use case diagram and activity diagram with Z language for formalization of library management system. **International Journal of Electrical and Computer Engineering**, v. 9, p. 3069, 2019.

[60] NIPKOW, T.; PAULSON, L. **Isabelle HOL-the tutorial 2001**.

[61] NIPKOW, T.; PAULSON, L. C.; WENZEL, M. **Isabelle/HOL**: A Proof Assistant for Higher-order Logic; Springer, 2002. v. 2283.

[62] NOULAMO, T.; TANYI, E.; NKENLIFACK, M.; LIENOU, J. P.; DJIMELI, A. Formalization Method of the UML Statechart by Transformation Toward Petri Nets. **IAENG International Journal of Computer Science**, v. 45, 2018.

[63] O'REGAN, G. Z formal specification language. *In*: **Concise Guide to Formal Methods**. Springer, 2017. p. 155–171.

[64] OMG. **OCL**: Object Constraint Language 2.4 2018. available at: https://www.omg.org/spec/OCL/2.4/PDF.

[65] OMG. **UML**: Unified Modeling Language, version 2.5, 2017. available at:https://www.omg.org/spec/UML/2.5.1/PDF.

[66] OMG. **XML Metadata Interchange (XMI) Specification 2.5.1 2015**. Available at: https://www. omg.org/spec/XMI/2.5.1/PDF.

[67] OUESLATI, R.; MOSBAHI, O.; KHALGUI, M.; LI, Z.; QU, T. Combining semi-formal and formal methods for the development of distributed reconfigurable control systems. **IEEE Access**, v. 6, p. 70426–70443, 2018.

[68] RIOBOO, R. Invariants for the FoCaL Language. **Annals of Mathematics and Artificial Intelligence**, v. 56, p. 273–296, 2009.

[69] SINGH, N. K.; LAWFORD, M.; MAIBAUM, T. S.; WASSYNG, A. A formal approach to rigorous development of critical systems. **Journal of Software: Evolution and Process**, p. e2334, 2021.

[70] SNOOK, C.; BUTLER, M. UML-B: Formal Modeling and Design Aided by UML. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, v. 15, p. 92–122, 2006.

[71] SNOOK, C.; SAVICKS, V.; BUTLER, M. Verification of UML Models by Translation to UML-B. *In*: **Proceedings of the Formal Methods for Components and Objects**. Springer, 2012. p. 251–266.

[72] SUNYÉ, G.; LE GUENNEC, A.; JÉZÉQUEL, J. M. Design Patterns Application in UML. *In*: **ECOOP 2000—Object-Oriented Programming**. Springer, 2000. p. 44–62.

[73] UNHELKAR, B. **Software Engineering with UML**. CRC Press, 2017.

[74] WAKRIME, A. A.; AYED, R. B.; COLLART-DUTILLEUL, S.; LEDRU, Y.; IDANI, A. Formalizing railway signaling system ERTMS/ETCS using UML/Event-B. *In*:. **Proceedings of the International Conference on Model and Data Engineering** Springer, 2018. p. 321–330.

[75] YANG, J. A framework for formalizing UML models with formal language rCOS. *In*: **Proceedings of the Frontier of Computer Science and Technology, 2009. FCST'09. Fourth International Conference on**. IEEE, 2009. p. 408–416.

[76] YANG, M.; Michaelson, G.; Pooley, R. Formal Action Semantics for a UML Action Language. **Journal of Universal Computer Science**, v. 14, p. 3608–3624, 2008.