software

Article

The MESSIR Flexible Scientific Approach to Requirements Engineering

Nicolas Guelfi

Faculty of Sciences, Technology and Medicine, Department of Computer Science, University of Luxembourg, L-4362 Esch-sur-Alzette, Luxembourg; nicolas.guelfi@uni.lu

Abstract: Among the software engineering development phases, requirements engineering is the one that has the most impact on project success or failure. To be executed in various contexts, there is an important need for flexibility and efficient tool support. A flexible requirements engineering method should include several levels allowing for more or less completeness and precision. Some project contexts would need a lightweight activity using structured natural language but still being guided and grounded partly on professional standards. Some more advanced projects would need more complete requirements documents and would benefit from a description language based on scientific notions allowing for better precision for specific system operations. Some business or safety critical systems would need an approach allowing for requirements simulation and verification. Requirements engineering education is an important objective to prepare future engineers to understand those requirements engineering needs and be prepared for practice in a professional setting. In the last five years, we have developed a requirements engineering method called **Messir** with a tool **Excalibur** and experiments in academia have been made to see how it was solving actual software engineering problems focusing first on requirements engineering education. Messir components represent in themselves some improvements w.r.t. the state of the art of the "standard" theories, methods and tools, mainly by introducing an improved requirements engineering process, language and verification support based on executable requirements specifications. Furthermore, the Messir approach solves also some actual problems related to software engineering education by offering a product line framework for setting up or improving courses in computer science curricula. The main result being to contribute to develop the software engineering capabilities of engineers and scientists that feed the job market in industry, research or education.

Keywords: requirements analysis; model-driven software engineering; software engineering environments; software engineering education

Citation: Guelfi, N. The MESSIR Flexible Scientific Approach to Requirements Engineering. *Software* **2022**, *1*, 80–106. https://doi.org/10.3390/software1010005

Academic Editor: Claus Pahl

Received: 29 January 2022
Accepted: 25 February 2022
Published: 7 March 2022

1. Introduction

Since the software crisis, as addressed in the NATO conference [1], a huge amount of progress has been made on the research, engineering and teaching dimensions. It is admitted that software engineering (SE) can be defined as "the disciplined application of engineering, scientific, and mathematical principles, methods, and tools to the economical production of quality software" [2]. In order to be more precise on what SE is, an important standardization effort by ISO and IEEE has been made in the last decade offering a proposal to define the SE body of knowledge [3]. This body of knowledge defines 15 knowledge areas (KA) decomposed into topics and sub-topics (as illustrated in Figures 1–4). The SWEBOK represents one of the best joint efforts from research, education and industry actors to improve the SE domain. It is thus used as a basis and is exploited in the remaining parts of the paper.

It is a fact that the expansion speed of the software development domain at conceptual and technological levels is so high that it is very difficult for researchers to produce SE knowledge solving SE problems such that they can be scaled and transferred in time to be deployed consistently in education and in real industrial projects.

The spectrum of SE approaches can be observed using the two following axis: science and engineering. Approaches at the extreme end of the science axis will only consider an SE artifact if it has a sound (rigorously defined in a complete and consistent way) mathematical basis. On the extreme of the other axis, one will consider only notions that proved to solve efficiently (time and money) actual problems encountered in real world projects considering the reality of technological and human resources.

A wide spectrum of approaches exists that we could see as going from so-called *formal or mathematical methods* such as all the ones that are based on or extend the fundamental ones including: *B* [4], *Z* [5], *VDM* [6], *Process Algebras* [7], or *Petri nets* [8]; through semi-formal methods such as the one proposed on the basis of *UML* [9] and its process *RUP* [10] or similar ones empowered by the model-driven engineering community (MDE) actively supported by the *OMG* including the method for development processes (i.e., "*agile methods*"), such as Scrum [11] or XP [12]; and finally, from in-house "spontaneous, intuitive and empirical" approaches to software development.

Software and hardware systems (i.e., IT systems) development is an engineering discipline for which we expect it to benefit from scientific methods. Unfortunately, if we analyze the success of formal methods [13], the success of agile methods [14] and the maturity of the IT industry in their project development [15], we can notice that scientific approaches did not penetrate *widely* the SE industry mostly due to the lack of economically viable, mature, pragmatic and ready to use solutions.

Among all the phases of software development, the requirements engineering activity has a high potential to impact the quality and prize of IT systems [16]. Requirements of good quality and well-managed projects allows to predict the project success in more than 90% of the cases [17].

The problem addressed by **MESSIR**, introduced in this paper, is to propose a requirements engineering approach that has variable coverage levels of science and engineering, as advocated by Ivar Jacobson et al. in [18], which is driven by pragmatics and that integrates sound theories, methods and tools. In addition, we aim to provide artifacts that can be deployed in SE curricula or in life-long learning trainings in order to be used by engineers to improve their requirements engineering expertise.

Being flexible for a requirements engineering approach means to have the possibility to describe requirements with a variable level of completeness and precision and still being able to move from one level to another one depending on the project context. Being efficient means to have means to execute faster and better the requirements analysis phase.

The main contributions contained in the **MESSIR** approach presented in this paper are:

- An integrated and flexible method for scientific requirements engineering;
- An approach supported by a software engineering environment;
- A flexible requirements textual description language;
- An improved use case modeling approach;
- An axiomatic and operational semantics supporting declarative executable requirements specifications necessary for automated verification.

Section 2 presents the background from software engineering in research, industry and education on which **MESSIR** is based and situates and compares our approach with the existing ones that share common goals; Section 3 presents in details the **MESSIR** requirements engineering approach including the process and models, its **EXCALIBUR** tool support and its application in software engineering education; Section 4 discusses the experimental and informal assessment made that is a basis for a more rigorous one; and Section 5 synthesizes the main contributions of the approach proposed and highlights interesting perspectives that will be developed in the near future.

Nb.	Knowledge Area Names
1	Software Requirements
2	Software Design
3	Software Construction
4	Software Testing
5	Software Maintenance
6	Software Configuration Management
7	Software Engineering Management
8	Software Engineering Process
9	Software Engineering Models and Methods
10	Software Quality
11	Software Engineering Professional Practice
12	Software Engineering Economics
13	Computing Foundations
14	Mathematical Foundations
15	Engineering Foundations

Figure 1. Swebok knowledge areas.

Nb	Topic Names
1	Modeling
2	Types of Models
3	Analysis of Models
4	Software Engineering Methods

Figure 2. Swebok topics for (9) software engineering models and methods.

Nb	SubTopic Names
1	Modeling Principles
2	Properties and Expression of Models
3	Syntax, Semantics, and Pragmatics
4	Preconditions, Postconditions, and Invariants

Figure 3. Swebok sub-topics for (9.1) modeling.

Nb	SubTopic Names
1	Heuristic Methods
2	Formal Methods
3	Prototyping Methods
4	Agile Methods

Figure 4. Swebok sub-topics for the (9.4) software engineering methods topic.

2. Background and State of the Art

2.1. Background

We introduce and motivate the existing theories, methods and tools available for software engineering that constitute the main background of **MESSIR**. Since the complexity resides in the selection, adaptation and integration of SE artifacts, we introduce and motivate the selected artifacts.

The basic notions issued from scientific research that represents milestones and on which **MESSIR** is based are the following main notions:

- **Theories**: Concerning basic or more advanced theoretical notions applied to SE and exploited in the context of formal methods, **MESSIR** is mainly grounded on the following ones:
  - (a) Set theory: concepts necessary for modeling information;
  - (b) Mathematical logic: basic concept for declarative characterization of information and behavior properties;
  - (c) Language theory: notions for textual modeling using domain specific languages;

   (d) Axiomatic semantics: notion for semantic interpretation of declarative descriptions;
   (e) Operational semantics: concepts for semantic interpretation of state modifications of abstract computing machine models.

- **Methods**: Modeling is a fundamental scientific activity which provides a Cartesian view of reality and is one of the main discovery and communication tools for the scientist. It implies to represent a phenomenon using a pre-defined modeling notation. Research has produced an important contribution in defining theories, methods and tools for modeling in all areas including computer science. For what concerns software engineering, model-driven engineering (MDE) has been intensively developed in the last 30 years mainly supported by the OMG around its model driven architecture initiative (MDA) [19]. In this context, it may be mentioned the following notions related to modeling for SE:
   (a) MOF: The Meta-Object-Facilities [20], providing a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems;
   (b) UML: The Unified Modeling Language [9], providing categories of modeling concepts adapted to model various types of properties of IT systems;
   (c) OCL: The Object Constraint Language [21], providing a formal language used to describe logical constraint expressions on UML models.
- **Tools**: Supporting SE activities with IT tools has often been a concern for researchers to support their SE solutions. In the context of SE tools, the problem is not only to find the correct set of tools supporting the targeted activities but to integrate them to set up the needed SE workbenches or environments. From this perspective, **MESSIR** considered the following general tools issued from research and industry:
   (a) Eclipse [22]: The open source integrated development environment having a powerful plug-in system for customization;
   (b) Xtext [23]: An advanced framework for development of general or domain specific languages;
   (c) Sirius [24]: A generic eclipse tool for graphical modeling workbench development;
   (d) Sictus Prolog [25]: A generic logic programming tool including constraints solver libraries allowing for implementation of axiomatic and operational semantics of declarative formal specifications.

For a requirements engineering approach to be pragmatic, it should also be grounded on industrial best practices. Determining the state of practice of software engineering in industry is not an easy task mainly due to the fact that the main goal of industry is neither to conduct such studies nor to make public their engineering practice. When they conduct such studies, they furthermore are not keen to publish them since they evolve in a highly competitive environment in which any information can become sensitive and may impact their business. Nevertheless, it is still possible to find direct or indirect ways to have some information on this subject.

According to the software engineering institute (SEI), "the quality of a system or product is highly influenced by the quality of the process used to develop and maintain it" [26]. CMMI-DEV is the last and most advanced maturity model proposed as a reference model covering activities, products and service development. It is thus interesting to observe industry practice through the capability maturity model in order to better determine the problem of improving software engineering practice. According to available studies [15,27–29], which we have witnessed during our own empirical experience in handling more than 600 internships in industry for bachelor and master students and as expert for the court of justice on conformance questions in trials for the software industry, we can state that:

(a) Approximately 3/4 of companies are at level 1 or 2 (initial or repeatable);
(b) Among the companies that go for CMMI appraisal, approximately 80% are at or below level 3 (defined) and 20% at level 4 (managed) and 5 (optimizing).

This meant for us that the focus should be made on problems encountered by low maturity level companies thus targeting requirements engineering and more precisely requirements definition, verification and validation.

The success that the so-called "agile methods" had, especially in companies having maturity level below 3, has also been studied. From our point of view, agile methods represent a "success story" from the software engineering point of view. This is because it penetrated industry at a high rate and contributed to improve the SE practices in industry, especially promoting the following SE knowledge areas: *Software Construction, Software Testing, Software Engineering Management, Software Engineering Process, Software Quality, Software Engineering Professional Practice*.

Available studies [30,31] or [32,33] show the significant adoption of the approach by the majority of the software development industry actors.

It can be deduced from this observation that important success factors for improving SE practice, mainly for those level 3 (or less) companies, are dependent on the following practices or facts:

(a) The development process should be iterative and incremental;
(b) Validation is mainly based on usage scenarios (user stories) as test cases;
(c) The customer should be easily integrated in the loop for requirement definition and validation;
(d) A large part of the product quality relies on the programmer who is the one-man band impacting all SE knowledge areas. A large project taskforce includes a collection of one-man bands that need adequate project management to be orchestrated;
(e) Documentation and modeling are seen as a burden even though partly tolerated.

In order to observe more precisely how modeling is used in industry, the following study is mentioned [34] which was made in a context of model-driven engineering projects, including over 17 companies and 250 engineers. The interesting facts that can be extracted from this study are:

(a) On modeling approaches used: 85% make use of UML, 40% use a DSL of their own design, 25% use a DSL provided by a tool, 25% use BPMN, 10% use SysML and MATLAB/Simulink;
(b) On UML Diagrams usage: 87% Class Diagram, 56% Activity Diagram, 38% use Case Diagram, 33% Sequence Diagram, 23% State Machine Diagram. Other diagrams, such as Component Diagram, Flow Diagram, Entity Relationship Diagram, Deployment Diagram, Object Diagram, Composite Structure Diagram, are rarely cited;
(c) On the perceived impact on (**P**)roductivity or (**M**)aintainability of Model-Driven Approaches used (it is indicated here the percentage of the respondents that declared to have noticed an impact on the indicated dimension): **communication**: 73.7% (P), 66.7% (M), use of models for **understanding** a problem at an abstract level: 73.4 (P), 72.2% (M), **code generation**: 67.8% (P), 56.9% (M), use of models to capture and **document** designs: 65.0% (P), 59.9 (M), use of model-to-model **transformations**: 50.8% (P), 42.6% (M), use of **domain-specific languages** (DSLs): 47.5% (P), 44.0% (M), model **simulation**—executable models: 41.7% (P), 39.4% (M), use of models in **testing**: 37.8% (P), 35.2% (M).

From those extracted facts, it has been deduced for **MESSIR** that:

(a) Modeling notations should be reduced to the smallest possible set and based on standard concepts;
(b) Design and usage of domain-specific languages is encouraged if based on standard concepts and supported by automated tools;
(c) Simulation, testing and code generation are mandatory features to make diagrams perceived as having impact on productivity;

(d) Usage of modeling notations should be flexible enough to allow various and freely chosen precision (i.e., scientific) levels.

To contribute to the maturity of requirements engineering and its diffusion, a requirements engineering approach should also be based on best educations practices and sound conceptual and terminological frameworks. To this aim, important milestones provided to the SE community are:

- SWEBOK—Software Engineering Body of Knowledge (submitted [3,35]);
- CS2013 Undergraduate Curriculum (submitted [36]);
- SE2014 Undergraduate Curriculum (submitted [37]);
- GSwE2009 Graduate Software Engineering 2009 ([38]).

In order to determine the actual coverage of the SWEBOK knowledge areas in education, the main results of evaluation made [39] over a sample of academic curricula (selected worldwide) are presented here, in order to determine how they were covering the SWEBOK knowledge areas (KAs). The sample has been structured using ISCED classification [40] and world regions defined for this study. The table given in Figure 5 shows the main global results giving, for each knowledge areas, the coverage percentages (mean, minimum, maximum and standard deviation) for the **18 curricula studied**.

KA	Name	Mean	Min	Max	StdDev
1	Software Requirements	71	33	100	24
2	Software Design	50	17	69	21
3	Software Construction	41	0	67	27
4	Software Testing	22	0	42	18
5	Software Maintenance	9	0	28	13
6	Software Configuration Management	4	0	17	7
7	Software Engineering Management	54	0	75	31
8	Software Engineering Process	28	0	40	16
9	Software Engineering Models and Methods	53	0	100	39
10	Software Quality	23	0	50	22
11	Software Engineering Professional Practice	57	0	84	34
12	Software Engineering Economics	20	0	60	24
13	Computing Foundations	46	0	79	38
14	Mathematical Foundations	42	0	71	32
15	Engineering Foundations	31	0	65	24

Figure 5. Global KA coverage (%).

This observation study has been useful to determine some of the main issues in SE education and their probable causes which largely impacted **MESSIR** goals and design. They can be summarized as follows:

(a) Most of the curricula consider software engineering as a discipline in itself and thus include a specific SE course only once in a full bachelor program. This makes the lecture difficult to design and execute since it implies that the teacher has to select a subset of the KAs to cover in a short time budget, and it also supposes that the teacher is expert in all the KAs which, of course, can rarely be the case;

(b) Most of the KA subtopics are covered thanks to Engineering Projects offered in the curriculum. Those projects are ideally used to cover KAs such as: Software Construction (3), Software Engineering Management (7), Software Engineering Professional Practice (11);

(c) Software Requirements (1) are quite well covered and often benefit from a full lecture;

(d) Software Testing (4) is less covered than expected, especially regarding its importance in industry;

(e) The importance, volume and coverage of theoretical computer science at bachelor level (being professional oriented or not) is way over what is described in the SWEBOK. This is mainly due to the profile of academic teachers that are mostly scientists, recruited based on scientific results that often have been obtained based on theoretical work. It also represents a wish and a need to provide a strong scientific and theoretical basis during education;

(f) The topics poorly addressed are Software Configuration Management (6), Software Engineering Economics (12) and Software Quality (29) and, more surprisingly, Software Maintenance (5). This last fact seems in direct opposition with the reality of engineering activities which, for the majority, consists in supporting software evolution and maintenance.

Furthermore, other studies [41] show that in the coming years, 74% of the jobs in the STEM (Science, Technology, Engineering and Mathematics) areas will be in the computer science area and only 30% of the new jobs in computer science could be fulfilled by newly graduated bachelor students.

The conclusion drawn is that it is mandatory to provide the education community with a knowledge transfer support that contributes to improve the capabilities of our engineers and technicians with regards to all the SWEBOK knowledge areas.

2.2. State of the Art

For what concerns the analysis about software engineering and its current practice in industry and education from a general perspective, the situation depicted in [42] presents a complete survey on requirements engineering practices and standardization needs which presents important principles that we share. For functional requirements documentation, it is necessary to support engineers in their current activities which focus on: textual structured requirements (42%); semi-formal use cases (39%), constraints expression (36%) and semi-formal data models (33%). It is shown that change management, traces between requirements design and code, and impact analysis are practiced and required. This study is confirmed by another interesting one given in [43] which confirmed the UML models choices, the necessary usage of textual description (i.e., which corresponds to the Messir documentation level) but also the priority management of requirements which is currently not yet supported in Messir.

In [44], the authors present an approach to SE education that supports the education of engineers for mastering modeling and validating models using jointly UML and OCL meta-models for requirements and design. It is also the chosen class diagram for concepts while the protocol of the system is partly modeled using state machine instead of protocol attributes as in **MESSIR**. Model execution is presented as a key feature for scientific development and the USE/UML SOIL language [45] is used. Sequence diagrams are used at design level for modeling objects interactions. Their internal survey about tools and modeling confirms the choices made which are shared by Messir concerning the models and tool support for SE education. It is worth mentioning that the choice of SWEBOK as a basis for defining the content of a software engineering curriculum has been made. On the knowledge acquisition results of using UML/OCL modeling notation and tools in requirements engineering education, the students' know-how results and perceptions we noticed in our experiences are similar to the ones made in the same context and described in [46]. Mainly, the fact that supporting tools are needed and that the use of OCL is tractable for engineers.

For what concerns a full textual first approach to requirements engineering, there is no existing work to our knowledge that is comparable to **MESSIR**. The majority of requirements engineering approaches that were developed combine requirements documentation standards such as the ISO/IEC/STANDARD IEEE 29148 [47], with some UML models, produced using a graphical syntax supported by a graphical editor, such as MagicDraw or Papyrus [48], which are manually integrated in the documentation. In addition to this, UML has been designed to be a generic modeling language targeting the modeling of the largest set of system properties at any abstraction level (analysis, design implementation). All the 13 sub-meta-models of UML allow for multiple ways to build semantically equivalent models. This capability makes UML not directly usable for well-guided requirements engineering methods but needs some selection and limitations to be used efficiently by engineers in a restricted time and knowledge context. This is the choice of methods, including **MESSIR**, developed with a shared spirit with the original Fusion method [49,50].

3. Results

In this section, the main elements are presented allowing us to understand the definition of the **MESSIR** approach for requirements engineering. Due to the limited size of this paper, it cannot be expected to have the full details of the approach, this is why the description is highly synthetic and focuses on the main characteristics. This allows a wide presentation of **MESSIR** showing the integration achieved. The full material that presents in more details the **MESSIR** methodology including the teaching material can be found in [51].

3.1. Running Example

MESSIR concepts illustrations are given using a crisis management case study. It is a simplified version of the one proposed in [52,53]. In the version used in this paper, the system (called *iCrash*) is intended to support the management of crisis situations. The *iCrash* stakeholders are:

- `Communication Companies`: They have the capacity to ensure communication of information between its customers and the *iCrash* system. Objectives are: to be able to deliver any SMS sent by any human to the *iCrash* 's dedicated phone number; to be able to transmit SMS messages from the ABC company that owns the *iCrash* system to any human having an SMS compatible device accessible using a phone number.
- `Humans`: A human is any person who considers himself related to a car crash (a specific type of crisis situation) either as a witness, a victim or an anonymous person. The objectives of a human are: to inform the *iCrash* system about the crisis situation he detected; to be sure that the ABC company has been informed about the situation; to be informed about the situation of the crisis he his related to as a victim or witness.
- `Coordinators`: Employees responsible for handling one or several crises. The objectives of a coordinator are: to securely monitor the existing alerts and crisis; to securely manage alerts and crisis until their termination.
- `Administrator`: The employee of the ABC company responsible for administrating the *iCrash* system. The objectives of an administrator are to add or delete coordinator actors from the system and its environment.
- `Creator`: Technician who is installing the system on the targeted deployment infrastructure. The objectives of a `Creator` are: to install the *iCrash* system; to define the values for the initial system's state; to define the values for the initial system's environment; to ensure the integration of the *iCrash* system with its initial environment.
- `Activator`: Logical representation of the active part of a system. It represents an implicit stakeholder belonging to the system's environment that interacts with the system autonomously without the need of an external entity. It is usually used for representing time triggered functionalities. The objectives of the *iCrash* `activator` are: to communicate the current time to the system; to notify the administrator that some crises are still pending for a too long time.

3.2. Messir Process

The **MESSIR** Requirements Engineering process (**MEVOP**) is **iterative** and **incremental** and depicted in Figure 6 which sketches the process activities. The main objective is to describe a set of functional and non-functional properties. The functional properties are modeled using five model types:

- Use case Model;
- Environment Model;

- Concept Model;
- Operation Model;
- Test Model.

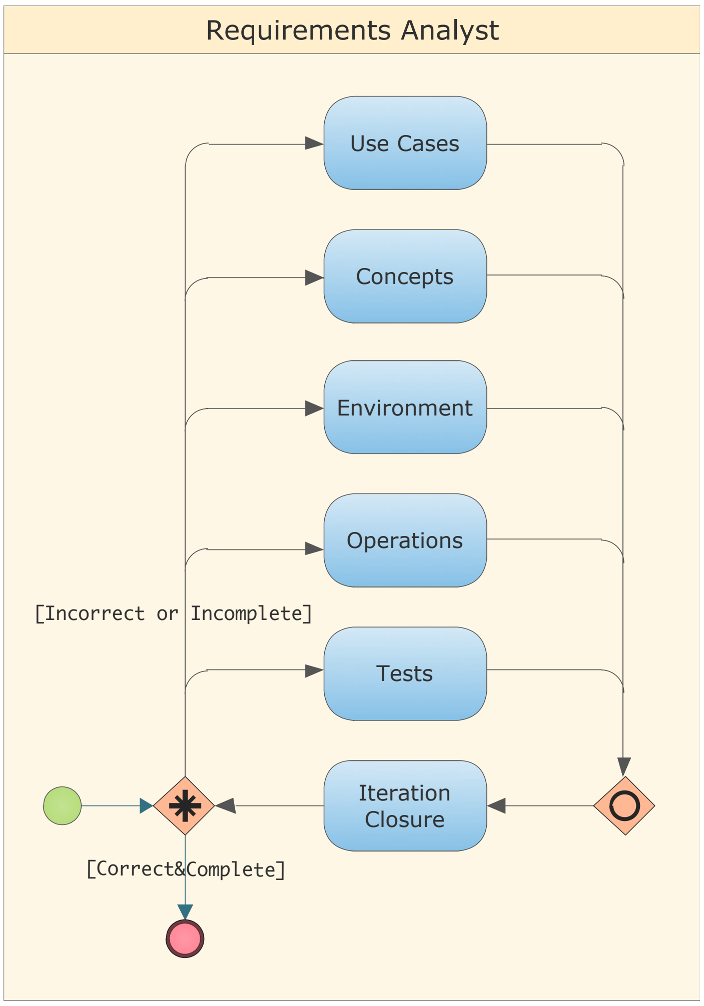


Figure 6. MESSIR MEVOP Diagram.

A requirements document can be provided using three precision levels which are:

- **Definition Level:** Using natural language descriptions (structured textual or graphical) for each of the predefined **MESSIR** model type;
- **Specification Level:** All what is provided at definition level plus the possibility to specify the system operation and tests using OCL language (i.e., object-oriented constraints specification of operations);
- **Simulation Level:** All what is provided at definition level plus the possibility to provide PROLOG executable code for operations and tests allowing for requirements simulation using the **MESSIR** simulator (**MESSIM**) in compliance with the **MESSIR** abstract machine **MESSAM**.

Figure 7 sketches the model focus by scientific level targeted. For example, it shows that if **Definition level** is targeted, then most of the focus will be on use case modeling.

Requirements descriptions are provided using textual languages and their use depends on the targeted scientific levels. Thus, it is up to the engineer to determine, for the requirements description, the scientific level targeted and the languages used. It can be chosen to use jointly different scientific levels for different functionalities of the system under consideration. Such choice is based on several project constraints and system characteristics. Safety or business critical parts of the system might need more precise and rigorous requirements descriptions, while the project delays, budget and team expertise might justify to provide only natural language descriptions of the requirements or, even, no description at all.

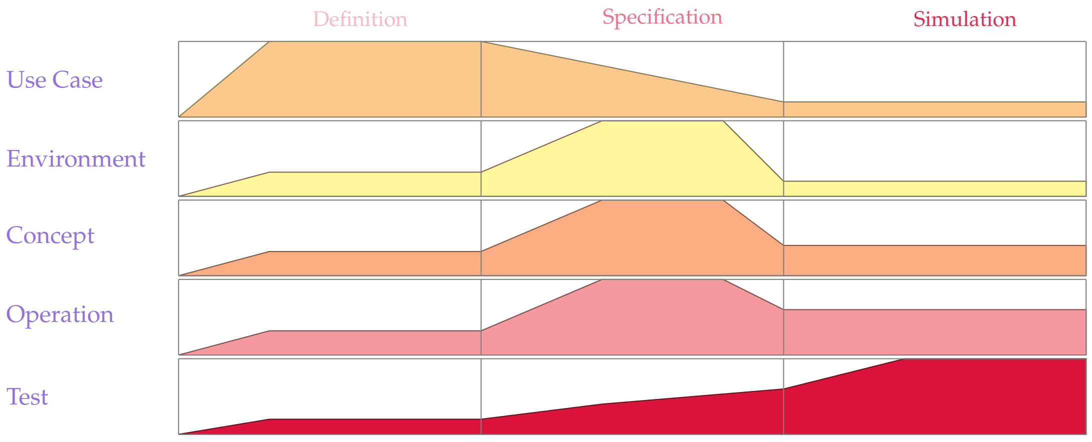


Figure 7. MESSIR MEVOP—Model Focus by Level.

MESSIR provides three domain-specific languages defined at meta-level using grammars supported by the Xtext technology [23]). The languages available are:

- **Documentation language (** `msrd`**)**: Allowing for free natural language descriptions but structured using domain-specific templates. Figure 18 provides an illustration of this template language for the system's operation `oeAddCoordinator` description. The generated documentation from this description is shown in Figure 19;
- **Specification language (** `msr`**)**: Allowing for more precise conceptual and logical descriptions given the use of the **MESSIR** language mcl which is an adapted executable version of the OCL language [21]. Figure 20 provides an extract illustrating this language for the system's operation `oeAddCoordinator` specification;
- **Axioperational Language (** `pl`**)**: A Prolog-compliant language is proposed to allow for ***axioperational*** (i.e., axiomatic and operational) semantics of the requirements descriptions. Figure 21 provides an extract illustrating this language for the same operation.

This approach allows the requirements analyst to tune the scientific level of the requirements description to the exact need and is synthesized in the generated documentation by a scientific level table using the system operations as observation criteria. The table shown in Figure 8 indicates the languages usage degree (+/−) by scientific level. The colors indicate the level (green, orange and red respectively for Definition, Specification and Simulation levels). Figure 9 shows this principle in the context of the *iCrash* case study. A "●" indicates the targeted level, a "✓" indicates a *reached* targeted level and, a " x" indicates a reached level. Thus, the targeted and reached levels are clearly indicated and the flexibility of the combinations covers pragmatically all the needs (i.e., reached level equal, lower or greater than targeted level).

Level	`msrd`	`msr`	`pl`
Definition	+++	+	−
Specification	++	+++	−
Simulation	+	+	+++

Figure 8. Categories of MESSIR languages by Scientific level.

Actor	Operation	Def.	Spec.	Sim.
actMsrCreator	oeCreateSystemAndEnvironment			x
actCoordinator	oeSetCrisisHandler	●	●	✓
	oeGetCrisisSet	x	●	
	oeGetAlertsSet		●	
	oeValidateAlert	●		
	oeInvalidateAlert	●		
	oeSetCrisisType	●		
	oeSetCrisisStatus	●		
	oeReportOnCrisis		●	
	oeCloseCrisis		✓	x
actAuthenticated	oeLogin	✓		
	oeLogout	✓		
actComCompany	oeAlert	x		●
actActivator	oeSollicitateCrisisHandling			●
	oeSetClock	●		
actAdministrator	oeAddCoordinator	●		
	oeDeleteCoordinator	●		

Figure 9. Illustration of **MESSIR** Scientific Levels Management for *iCrash* .

MESSIR includes a detailed process definition giving guidelines for activities to iteratively and incrementally engineer the requirements description models [54]. The different models to be provided are introduced in the next sections.

3.3. Requirements Engineering Basic Concepts

MESSIR considers that the system under development is made of a central system (called *system*) and its *environment*. The system has an observable state and a set of *system operations*. An environment is defined by a set of *actor* instances each of which having an *output interface* defining a set of asynchronous messages that can be sent by the actor to the system and an *input interface* defining a set of messages received by the actor from the system. Interactions are sequences of input and output messages between the environment and the system. An actor sends an output message in order to trigger a system's atomic and instantaneous operation which might include: change the system's state or the environment composition and generate message sending to the actors of the environment.

3.4. Messir Use Case Model

Use cases analysis is a very efficient technique for requirements elicitation which has been studied and applied efficiently in industry [55–57]. The main objective of use case modeling is to specify and illustrate business use cases of the actors concerned directly or not by the system under development. **MESSIR** introduces a new use case approach that overcomes some limitations encountered in practice when using tool-supported "standard" approaches. To this purpose, **MESSIR** use case analysis has the following characteristics:

- Uses a **textual** modeling language for engineering the use case model;
- Uses graphical use case diagrams generated from textual models as communication views. Views are based on the standard UML use case diagrams;
- Replaces `extends` and `includes` by a `reuse` association to simplify the modular description of use cases;
- Replaces sequential ordering of activities (`main success scenario` and `extensions`) by a set of steps and a set of ordering constraints over those steps. This solves a main issue in use case writing which is misuse or inefficient use of main scenario and extensions;
- Introduces use case instances to illustrate concrete scenarios that represent meaningful scenarios satisfying the use case model description;
- Restrict abstraction to three levels (summary, user-goal and sub-function);
- Introduces for each use case the possibility to describe typed parameters;

- Introduces for each actor the notions of role (primary/secondary), mode (direct/indirect), involvement (pro-active/active/reactive/passive) and multiplicity;
- Allows to provide documentation for each use case including: goal description and protocol/pre/post conditions description if necessary);
- Provides **graphical** representations of use cases and use cases instances using **MESSIR** use-case diagrams and sequence diagrams generated, maintained automatically w.r.t the textual descriptions. This contributes to make **MESSIR** useful for production and not only used as a documentation approach as often encountered.

Figure 10 provides a textual description of a use case of the **MESSIR** use case model using the `msr` language and Figure 11 represents the automatically generated diagrammatic view for this use case. Figure 12 shows a textual description extract of a use case instance and Figure 13 its graphical representation.

```
Use Case Model {
 use case system summary
  suGlobalCrisisHandling() {
  actor actCoordinator[primary,active]

  reuse ugSecurelyUseSystem[1..*]
  reuse ugMonitor[1..*]
  reuse ugManageCrisis[1..*]

  step a: actCoordinator
      executes ugSecurelyUseSystem
  step b: actCoordinator
      executes ugMonitor
  step c: actCoordinator
      executes ugManageCrisis

  ordering constraint
   "steps (a) (b) and (c) executions are interleaved
   (steps (b) and (c) have their protocol constrained by
  steps of (a))."
  ordering constraint
    "steps (a) (b) and (c) can be executed multiple times."
```

Figure 10. Illustration of a textual use case description.

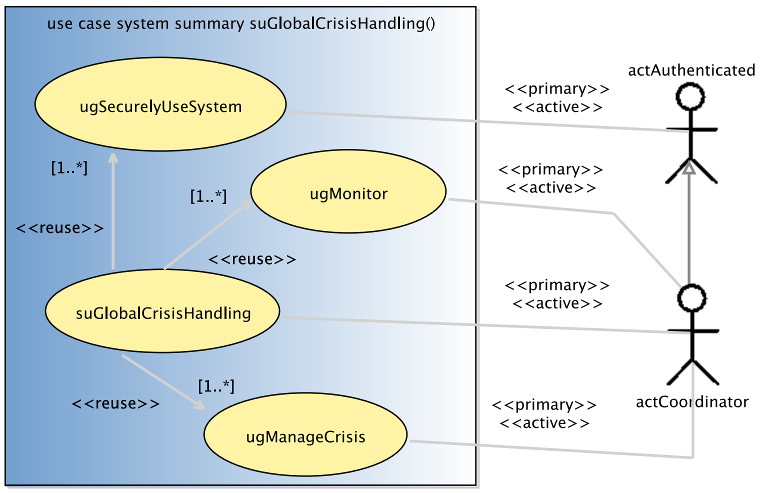


Figure 11. Illustration of generated graphical view for a use case description.

```
//------------------------------------------------------------
   steve
   executed instanceof subfunction
      oeLogin("steve","pwdMessirExcalibur2017"){
       ieMessage('You are logged ! Welcome ...') returned to
  steve
      }
//------------------------------------------------------------
   steve
   executed instanceof subfunction
      oeGetCrisisSet("pending"){
       ieSendACrisis("crisis with ID 1 details") returned to
  steve
      }
//------------------------------------------------------------
   steve
   executed instanceof subfunction
      oeSetCrisisHandler("1"){
       ieSmsSend("+3524666445252","The handling of your alert
   by our services is in progress !")
       returned to tango
       ieMessage("You are now considered as handling the
  crisis !")
       returned to steve
      }
//------------------------------------------------------------
```

Figure 12. Illustration of a textual use case instance description.

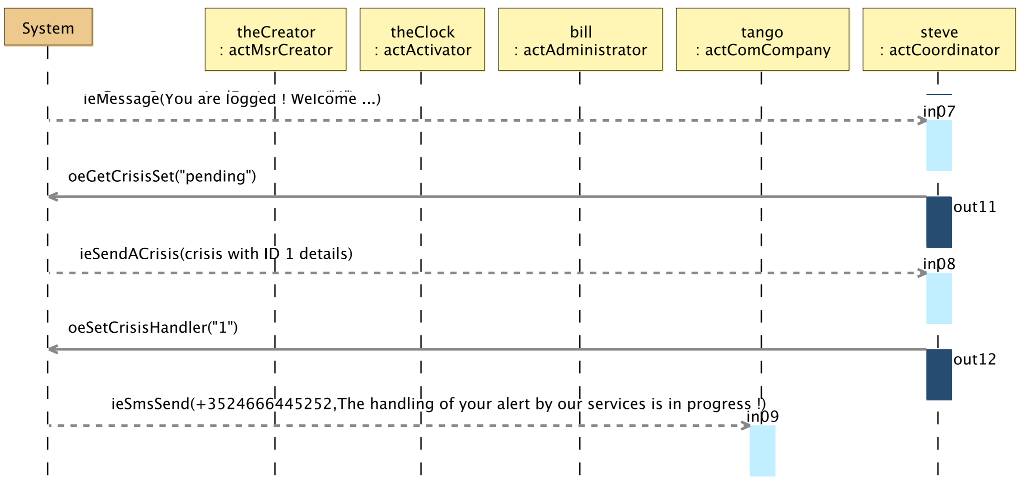

System
theCreator : actMsrCreator
theClock : actActivator
bill : actAdministrator
tango : actComCompany
steve : actCoordinator
ieMessage(You are logged ! Welcome ...)
in07
oeGetCrisisSet("pending")
out11
ieSendACrisis(crisis with ID 1 details)
in08
oeSetCrisisHandler("1")
out12
ieSmsSend(+3524666445252,The handling of your alert by our services is in progress !)
in09

Figure 13. Illustration of generated graphical view for a use case instance description.

3.5. Messir Environment Model

The objective of the environment model is to describe *actors* with their *output* and *input* interfaces used to interact with the system which is the same environment notion as the one introduced in [58]. The environment model is textually described using the **MESSIR** language and graphical view generated using simple class diagram notation.

Figure 14 illustrates such a textual description for a part of the *iCrash* example for the coordinator actor (`actCoordinator`) while Figure 15 provides its graphical view. An actor is associated to the system state root (instance of the `ctState` class) (the role provides the association name end) and can inherit from another actor (extends). It must be mentioned that the message parameters are typed using the concept model below. So, iterations and increments must be made consistently between the environment model and the concept model (the **EXCALIBUR** tool [59] provides support for this task). Those messages and their parameter types are partly elicited thanks to the use cases (sub-functions) and the use case instances.

```
actor actCoordinator
    role rnactCoordinator
    cardinality [0..*]
    extends actAuthenticated{

 operation init():ptBoolean

 output interface outactCoordinator{
  operation oeInvalidateAlert(AdtAlertID:dtAlertID ):
     ptBoolean
  operation oeCloseCrisis(AdtCrisisID:dtCrisisID ):ptBoolean
  operation oeGetAlertsSet(AetAlertStatus:etAlertStatus ):
     ptBoolean
  operation oeGetCrisisSet(AetCrisisStatus:etCrisisStatus ):
     ptBoolean
  operation oeSetCrisisHandler(AdtCrisisID:dtCrisisID ):
     ptBoolean
```

• • •

```
 input interface inactCoordinator{
  operation ieSendAnAlert(ActAlert:ctAlert ):ptBoolean
  operation ieSendACrisis(ActCrisis:ctCrisis ):ptBoolean
  }
}
```

Figure 14. Illustration of a textual Environment Model description.

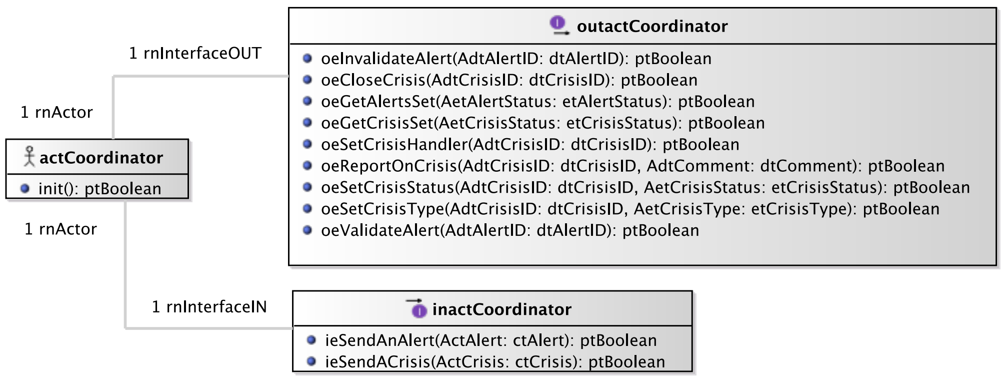

1 rnInterfaceOUT
1 rnActor
actCoordinator
init(): ptBoolean
1 rnActor
1 rnInterfaceIN
outactCoordinator
oeInvalidateAlert(AdtAlertID: dtAlertID): ptBoolean
oeCloseCrisis(AdtCrisisID: dtCrisisID): ptBoolean
oeGetAlertsSet(AetAlertStatus: etAlertStatus): ptBoolean
oeGetCrisisSet(AetCrisisStatus: etCrisisStatus): ptBoolean
oeSetCrisisHandler(AdtCrisisID: dtCrisisID): ptBoolean
oeReportOnCrisis(AdtCrisisID: dtCrisisID, AdtComment: dtComment): ptBoolean
oeSetCrisisStatus(AdtCrisisID: dtCrisisID, AetCrisisStatus: etCrisisStatus): ptBoolean
oeSetCrisisType(AdtCrisisID: dtCrisisID, AetCrisisType: etCrisisType): ptBoolean
oeValidateAlert(AdtAlertID: dtAlertID): ptBoolean
inactCoordinator
ieSendAnAlert(ActAlert: ctAlert): ptBoolean
ieSendACrisis(ActCrisis: ctCrisis): ptBoolean

Figure 15. Illustration of the generated graphical view for a *Environment Model* description.

3.6. Messir Concept Model

The objective is to specify the types structuring the information either used to define the system's state during its life cycle or exchanged with its environment. The **MESSIR** Concept Model provides the specification of: **Primary types** used for structuring the information defining the system's state; **Secondary types** for additional types necessary either for the information used in input or output message parameters, for actors' attributes or for the operation specifications given in the operation model.

Primary and secondary types can be defined using the following meta-types: *Class type*, *DataType type* and *Relation type* consistent with the standard data structures:

- A class type is defined using a name, a set of typed attributes and an optional supertype name. There is only one class type named `ctState` dedicated to the system state root. A class attribute type must be of datatype type. Each primary class type is associated to the `ctState` class by an aggregation association in order to provide an access to instances from the `ctState` instance necessary to describe operation semantics (see below);

- A datatype type can be a *structured data type*, an *enumeration* or a *primitive type*. A *structured data type* defines a set of tuples whose elements (named attributes) are typed using any *primary type*. Primitive types are: `Boolean`, `Integer`, `Real` or `String` (a naming convention in **MESSIR** proposes to use prefixes to ease understanding while reading textual descriptions (e.g., `pt,dt,ct,et,oe,ie` will stand for primitive type, datatype, class type, enumeration type, output event, input event) and correspond to types defining atomic values supported by the abstract machine of the simulator (all the useful primitive type operations are provided in **MESSIR** libraries written in **MESSIR** at simulation level);
- Relation types are aggregation, composition or (common) association;
- Functions can be part of a class type definition or a datatype type definition. Those operations semantically correspond to logical predicates and are used to allow for a structured and concise writing of predicates.

Figure 16 illustrates such a textual description for the concept model using a class-type primary type `ctAlert`. Its graphical representation is shown in Figure 17 which depicts the full concept model for the *iCrash* example.

```
Concept Model {

 Primary Types{
```

• • •

```
  class ctAlert role rnctAlert cardinality [0..*]{
   attribute id:dtAlertID
   attribute status: etAlertStatus
   attribute location:dtGPSLocation
   attribute instant:dtDateAndTime
   attribute comment:dtComment

   operation init(    Aid:dtAlertID ,
         Astatus:etAlertStatus ,
         Alocation:dtGPSLocation ,
         Ainstant:dtDateAndTime ,
         Acomment:dtComment ):ptBoolean
   operation isSentToCoordinator(AactCoordinator:
      actCoordinator ):ptBoolean

  }
```

Figure 16. Illustration of a textual *Concept Model* description.

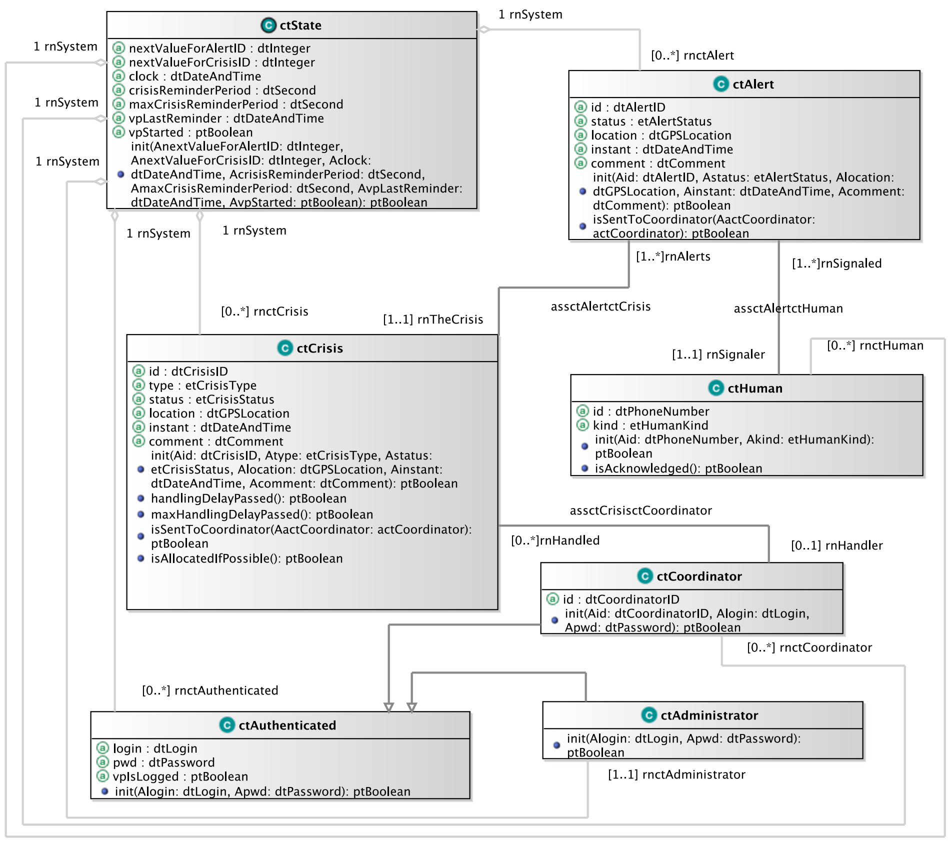

ctState
nextValueForAlertID : dtInteger
nextValueForCrisisID : dtInteger
clock : dtDateAndTime
crisisReminderPeriod : dtSecond
maxCrisisReminderPeriod : dtSecond
vpLastReminder : dtDateAndTime
vpStarted : ptBoolean
init(AnextValueForAlertID: dtInteger, AnextValueForCrisisID: dtInteger, Aclock: dtDateAndTime, AcrisisReminderPeriod: dtSecond, AmaxCrisisReminderPeriod: dtSecond, AvpLastReminder: dtDateAndTime, AvpStarted: ptBoolean): ptBoolean
1 rnSystem
1 rnSystem
1 rnSystem
1 rnSystem
1 rnSystem
1 rnSystem
[0..*] rnctAlert
ctAlert
id : dtAlertID
status : etAlertStatus
location : dtGPSLocation
instant : dtDateAndTime
comment : dtComment
init(Aid: dtAlertID, Astatus: etAlertStatus, Alocation: dtGPSLocation, Ainstant: dtDateAndTime, Acomment: dtComment): ptBoolean
isSentToCoordinator(AactCoordinator: actCoordinator): ptBoolean
[1..*]rnAlerts
[1..*]rnSignaled
assctAlertctCrisis
assctAlertctHuman
[0..*] rnctCrisis
[1..1] rnTheCrisis
[1..1] rnSignaler
[0..*] rnctHuman
ctCrisis
id : dtCrisisID
type : etCrisisType
status : etCrisisStatus
location : dtGPSLocation
instant : dtDateAndTime
comment : dtComment
init(Aid: dtCrisisID, Atype: etCrisisType, Astatus: etCrisisStatus, Alocation: dtGPSLocation, Ainstant: dtDateAndTime, Acomment: dtComment): ptBoolean
handlingDelayPassed(): ptBoolean
maxHandlingDelayPassed(): ptBoolean
isSentToCoordinator(AactCoordinator: actCoordinator): ptBoolean
isAllocatedIfPossible(): ptBoolean
ctHuman
id : dtPhoneNumber
kind : etHumanKind
init(Aid: dtPhoneNumber, Akind: etHumanKind): ptBoolean
isAcknowledged(): ptBoolean
assctCrisisctCoordinator
[0..*]rnHandled
[0..1] rnHandler
ctCoordinator
id : dtCoordinatorID
init(Aid: dtCoordinatorID, Alogin: dtLogin, Apwd: dtPassword): ptBoolean
[0..*] rnctCoordinator
[0..*] rnctAuthenticated
ctAuthenticated
login : dtLogin
pwd : dtPassword
vpIsLogged : ptBoolean
init(Alogin: dtLogin, Apwd: dtPassword): ptBoolean
ctAdministrator
init(Alogin: dtLogin, Apwd: dtPassword): ptBoolean
[1..1] rnctAdministrator

Figure 17. Illustration of the generated graphical view for a *Concept Model* description.

3.7. Messir Operation Model

The objective of operation specification is to provide the *properties* that "characterize" all valid *"executions"* for each system operation. Considering a transition system as semantic model, we need to characterize the transitions $< state_i, env_i > \xrightarrow{oe_i,[ie_1,\ldots,ie_n]} < state_{i+1}, env_{i+1} >$ in which $< state_i, env_i >$ (resp. $< state_{i+1}, env_{i+1} >$) represents the system's state and environment state **@pre** (resp. **@post**), oe_i the system operation executed triggered by an output event sent by an actor, ie_i the input events sent to actors that define the specification semantics.

An operation specification in **MESSIR** is an axiomatic specification of its semantic properties. Those properties are grouped in the following categories: *Pre-protocol*, *Pre-functional*, *Post-functional and Post-Protocol*:

- **Pre-functional properties** are conditions under which the system operation is considered *defined*. Expressed using: the system's state **@pre**, the environment state **@pre** and the operation parameters;
- **Pre-protocol properties** are conditions under which the operation is considered *available*. Expressed using: the system's state **@pre** including the additional variables for protocol specification, the environment state **@pre** and the operation parameters;
- **Post-functional properties** are conditions describing the operation's *functionality*. It characterizes a set of valid post-states for the system and the environment, a multiset of messages sent to actor instances. Expressed using: the system's state **@pre** or **@post**, the environment state **@pre** or **@post** and the operation parameters;
- **Post-protocol properties** are conditions describing the operation's *impact* on the system's interaction *protocol* (i.e., the system status **@post** makes available only the wished operations). They are expressed using the variables for protocol specification **@post**.

To illustrate the operation model, we consider an operation used by the administrator actor to add a coordinator actor. Figure 18 provides its description at **definition level** using the `msrd` documentation grammar of **MESSIR**. Figure 19 illustrates the automatically generated presentation of this definition level documentation. Figure 20 provides its description at **specification level** using the `msr` specification grammar of **MESSIR** which includes the "OCL -like" ocl constraints specification of the conditions. Finally, Figure 21 shows an extract of its implementation using the Prolog language (this can only be done by requirements engineers having a high scientific level) and using the **MESSIR** Prolog layer offered by the **MESSIM** simulator (e.g. **msrOp, msrNav** predicates).

```
@@Operation
icrash.environment.actAdministrator.outactAdministrator.
    oeAddCoordinator
```

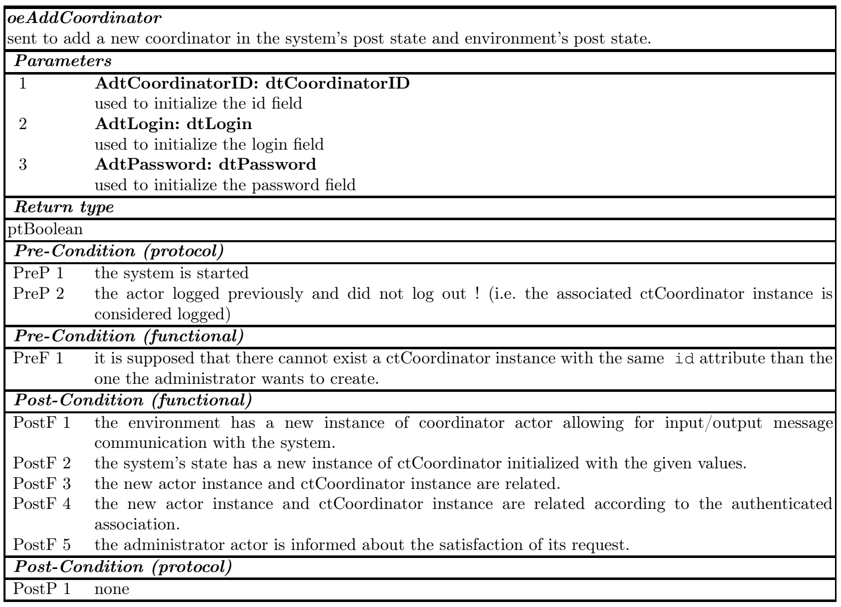
```
@description
"sent to add a new coordinator in the system's post state and
    environment's post state."

//preProtocol descriptions
@preP
"the system is started"
@preP
"the actor logged previously and did not log out ! (i.e. the
    associated ctAdministrator instance is considered logged)"
@endPreP

//preFunctional descriptions
```

```
@postF
"the system's state has a new instance of ctCoordinator
    initialized with the given values."
@postF
"the new actor instance and ctCoordinator instance are related
    ."
```

Figure 18. Illustration of a generated documentation for an Operation description at **Definition level**.



oeAddCoordinator	
sent to add a new coordinator in the system's post state and environment's post state.	
Parameters	
1	**AdtCoordinatorID: dtCoordinatorID** used to initialize the id field
2	**AdtLogin: dtLogin** used to initialize the login field
3	**AdtPassword: dtPassword** used to initialize the password field
Return type	
ptBoolean	
Pre-Condition (protocol)	
PreP 1	the system is started
PreP 2	the actor logged previously and did not log out ! (i.e. the associated ctCoordinator instance is considered logged)
Pre-Condition (functional)	
PreF 1	it is supposed that there cannot exist a ctCoordinator instance with the same `id` attribute than the one the administrator wants to create.
Post-Condition (functional)	
PostF 1	the environment has a new instance of coordinator actor allowing for input/output message communication with the system.
PostF 2	the system's state has a new instance of ctCoordinator initialized with the given values.
PostF 3	the new actor instance and ctCoordinator instance are related.
PostF 4	the new actor instance and ctCoordinator instance are related according to the authenticated association.
PostF 5	the administrator actor is informed about the satisfaction of its request.
Post-Condition (protocol)	
PostP 1	none

Figure 19. Illustration of an Operation description documentation table at **Definition level**.

```
operation: actAdministrator.outactAdministrator.
    oeAddCoordinator(AdtCoordinatorID:dtCoordinatorID, AdtLogin
    :dtLogin, AdtPassword:dtPassword):ptBoolean
```

```
postF{
 let TheSystem: ctState in
 let TheactCoordinator:actCoordinator in
 let ThectCoordinator:ctCoordinator in
 self.rnActor.rnSystem = TheSystem
 and self.rnActor = TheActor
/* PostF01 */
 TheactCoordinator.init()
/* PostF02 */
 and ThectCoordinator.init(AdtCoordinatorID,AdtLogin,
    AdtPassword)

/* PostF03 */
 and TheactCoordinator@post.rnctCoordinator = ThectCoordinator

/* PostF04 */
 and ThectCoordinator@post.rnactAuthenticated =
    TheactCoordinator

/* PostF05 */
 and TheActor.rnInterfaceIN^ieCoordinatorAdded()
```

Figure 20. Illustration of a Specification level operation description.

```
msrop(outactAdministrator,
   oeAddCoordinator,
   [preProtocol,Self,
    AdtCoordinatorID,
    AdtLogin,
    AdtPassword
    ],
   []):-
```

• • •

```
/* PostF03 */
 msrNav([TheactCoordinator],
     [msmAtPost,rnctCoordinator],
     [ThectCoordinator]),
/* PostF05 */
  msrNav([TheActor],
     [rnInterfaceIN,
      ieCoordinatorAdded,[]],
     [[ptBoolean,true]]),
```

Figure 21. Illustration of a Simulation level operation description.

3.8. *Messir Test Model*

The objective of the test model is to define test cases for verification and validation purposes. A test case is a sequence of test steps. Each test step defines: the *test message* characterizing the system operation triggered; the *test constraints* which define the domain for the parameter values and the *test oracle* to define the test step acceptance conditions.

MESSIR mainly exploits test cases to:

- Verify the requirement descriptions by simulation to detect if the specification contains errors;

- Validate the requirements description by asking the system's customer to determine validation test cases;
- Verify the delivered *system program* produced after each project implementation iteration by using the test case specification to implement verification test cases to find errors in the implemented program.

Figure 22 illustrates a textual description for a test step of a test case for the *iCrash* system. This test step is given at specification level and tests the correct execution of the operation `oeSetCrisisHandler` triggered by the coordinator actor. The test step is considered successful if the simulator can execute a valid transition on the abstract machine and if the oracle constraint is satisfied. Figure 23 illustrates a graphical representation using UML sequence diagram generated from an actual test case instance either produced by the simulator or manually provided, to illustrate the expected execution of the test case.

```
test step ts12oeSetCrisisHandler order 12{
   variables{
     TheActor : actCoordinator
     AdtCrisisID : dtCrisisID
    }
   constraints{
     TheActor=TheSystem.rnactCoordinator
     ->select(a | a.rnctCoordinator.login.value.eq('steve'))
     ->any2(true)
     and AdtCrisisID.value= '1'
    }
   test message{
    out:TheActor sends to system actCoordinator.
  outactCoordinator.oeSetCrisisHandler(AdtCrisisID)
   }
   oracle{
    variables{
     AMessage:ptString
     AdtPhoneNumber:dtPhoneNumber
     AdtSMS:dtSMS
     ActAlert:ctAlert

     TheComCompany: actComCompany
     TheCoordinator:actCoordinator
    }
    constraints{
     AMessage = 'You are now considered as handling the
  crisis !'
     AdtSMS.value = 'The handling of your alert by our
  services is in progress !'
     TheComCompany.inactComCompany.ieSmsSend(AdtPhoneNumber,
  AdtSMS)
     TheCoordinator.inactCoordinator.ieSendAnAlert(ActAlert)
     TheActor.inactAuthenticated.ieMessage(AMessage)
   }
  }
  }
//
```

Figure 22. Illustration of a Specification level test description.

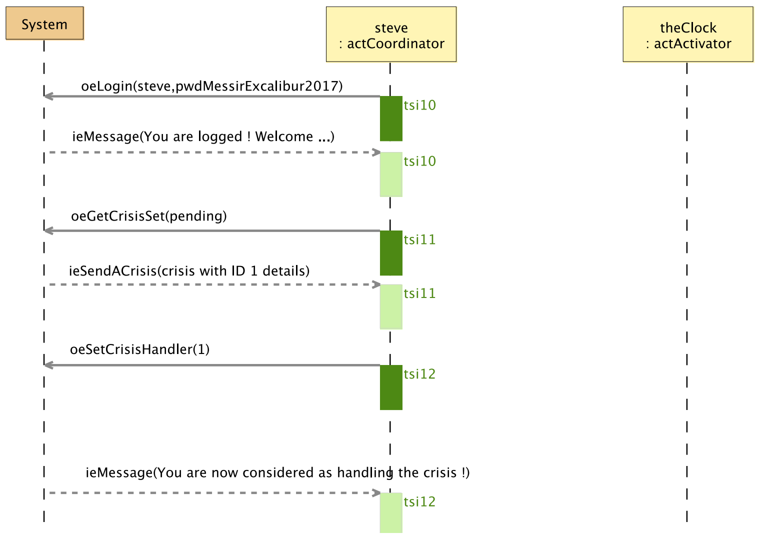

System
steve
: actCoordinator
theClock
: actActivator
oeLogin(steve,pwdMessirExcalibur2017)
tsi10
ieMessage(You are logged ! Welcome ...)
tsi10
oeGetCrisisSet(pending)
tsi11
ieSendACrisis(crisis with ID 1 details)
tsi11
oeSetCrisisHandler(1)
tsi12
ieMessage(You are now considered as handling the crisis !)
tsi12

Figure 23. Illustration of a generated graphical representation of a *Test instance* description.

3.9. Non-Functional Requirements

MESSIR bases the requirements description for non-functional requirements (NFR) on the ISO standard on quality requirements [60] (SQUARE) including a quality model organized in *Quality Criteria* (see Figure 24) and Quality sub-characteristics (such as Time behavior, Resource and utilization Capacity for the performance-efficiency quality criteria).

1	**Functional suitability**
2	**Performance efficiency**
3	**Compatibility**
4	**Usability**
5	**Reliability**
6	**Security**
7	**Maintainability**
8	**Portability**

Figure 24. SQUARE Main Quality Criteria.

Currently, **MESSIR** only allows for inclusion of functional requirements using natural language based on the ISO standard structure. It is planned to integrate a domain-specific language for NFRs that will allow for better handling of NFRs improved with some quantification criteria based on the ISO standard and the formal framework proposed in [61].

3.10. Messir Tool Support

Software engineering is the pragmatic integration of theories, methods and tools for software production. The **MESSIR** approach is supported by a software engineering environment that is built using, as a central part, its specifically designed SE workbench **EXCALIBUR** ([59]).

A software engineering environment set up for a software development project aims at supporting the following main dimensions: **Requirements Analysis**, **Design**, **Construction**, **Management**, **Quality** and **Maintenance**.

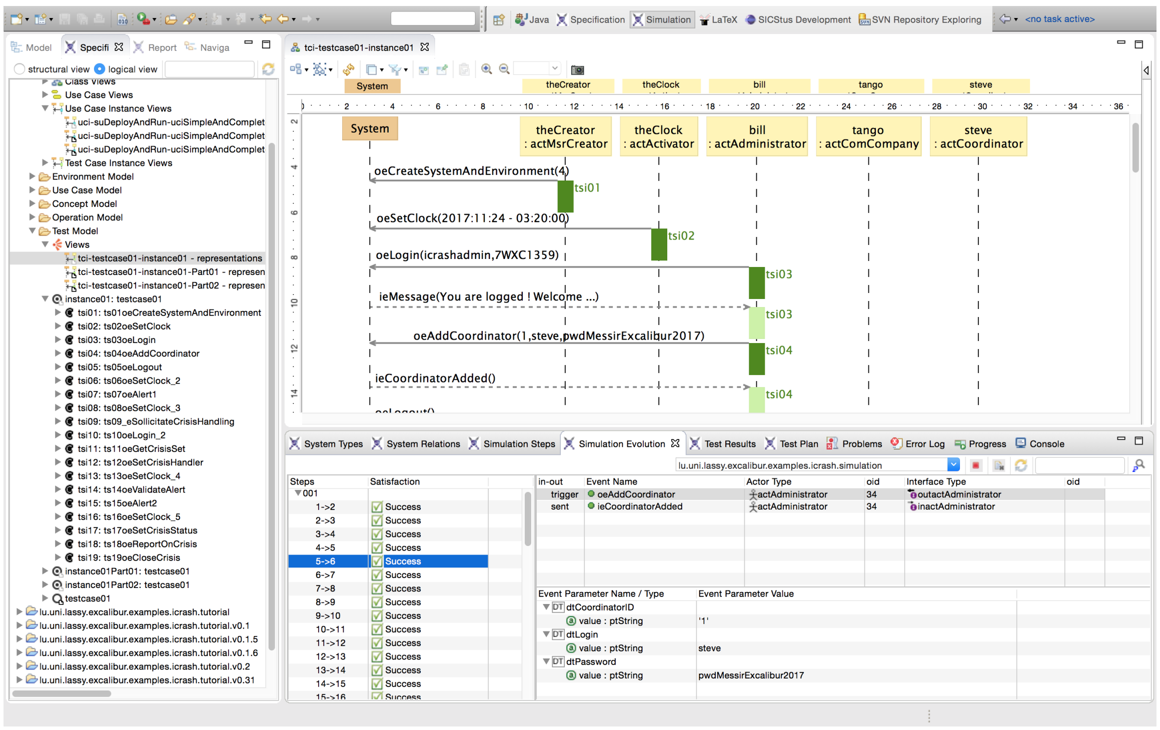
We have designed and developed an Eclipse plug-in (called **EXCALIBUR**) to support the requirements engineering as defined in the **MESSIR** methodology.

MESSIR Requirements Analysis is supported by the **EXCALIBUR** SE workbench allowing for the requirements' textual and graphical modeling, the verification using partial Prolog code and for the complete formatted report generation. Technologies used are: Eclipse, Sirius, Xtext, UML, OCL, Prolog, Latex, PDF.

EXCALIBUR was presented at the tools session of the 11th ACM SIGPLAN International Conference on Software Language Engineering [62]. In this short paper it is included a related work section in which the differences between Excalibur and comparable approaches can be found. Figure 25 shows a sample of the tool interface during simulation of test case instances provided textually and graphically abstracted using a sequence diagram. The lower panel allows to monitor the test results as well as the **MESSAM** state machine state observable at each test step.

In order to allow for wider software engineering support, in the context of our case study *iCrash* , we have an integrated eclipse plugin that can also be used outside the scope of **MESSIR** to support the following activities:

- **Design:** Eclipse UML Designer for production of design graphical models, design document using the provided template for Latex. Technologies used are: UML, Latex, PDF;
- **Construction:** Java for functionalities, JavaFx for graphical user interfaces, MySQL for data persistence, Java RMI for distributed processing, Apache, Tomcat and qmail for internet services. Technologies used are: Eclipse, Java, JavaFx, efxclipse, Apache, TomCat, MySQL, qmail;
- **Management:** Atlassian Confluence for knowledge base collaboration tool, SubVersioN versioning and revision control tool, Atlassian Bamboo and Apache Maven continuous integration server for project build Technologies used are: Atlassian, Confluence, Bamboo, SVN, Maven;
- **Quality:** Test-based Verification & Validation using the Prolog simulator, the SWTbot testing tool for graphical user interface, JUnit unit testing framework and EclEmma Java code coverage. Ensuring syntax validation tools using Eclipse and Xtext frameworks. Technologies used are: Eclipse, Xtext, Prolog, Junit, SWTbot;
- **Maintenance:** Atlassian JIRA as issue tracking tool, SubVersioN for versioning and revision control tool, Atlassian Bamboo and Apache Maven continuous integration server for project builds. Technologies used are: Atlassian, Jira, Bamboo, Maven.



```
Class Views
Use Case Views
Use Case Instance Views
    uci-suDeployAndRun-uciSimpleAndComplet
    uci-suDeployAndRun-uciSimpleAndComplet
    uci-suDeployAndRun-uciSimpleAndComplet
Test Case Instance Views
Environment Model
Use Case Model
Concept Model
Operation Model
Test Model
  Views
    tci-testcase01-instance01 - representations
    tci-testcase01-instance01-Part01 - represen
    tci-testcase01-instance01-Part02 - represen
  instance01: testcase01
    tsi01: ts01oeCreateSystemAndEnvironment
    tsi02: ts02oeSetClock
    tsi03: ts03oeLogin
    tsi04: ts04oeAddCoordinator
    tsi05: ts05oeLogout
    tsi06: ts06oeSetClock_2
    tsi07: ts07oeAlert1
    tsi08: ts08oeSetClock_3
    tsi09: ts09_eSollicitateCrisisHandling
    tsi10: ts10oeLogin_2
    tsi11: ts11oeGetCrisisSet
    tsi12: ts12oeSetCrisisHandler
    tsi13: ts13oeSetClock_4
    tsi14: ts14oeValidateAlert
    tsi15: ts15oeAlert2
    tsi16: ts16oeSetClock_5
    tsi17: ts17oeSetCrisisStatus
    tsi18: ts18oeReportOnCrisis
    tsi19: ts19oeCloseCrisis
  instance01Part01: testcase01
  instance01Part02: testcase01
  testcase01
lu.uni.lassy.excalibur.examples.icrash.tutorial
lu.uni.lassy.excalibur.examples.icrash.tutorial.v0.1
lu.uni.lassy.excalibur.examples.icrash.tutorial.v0.1.5
lu.uni.lassy.excalibur.examples.icrash.tutorial.v0.1.6
lu.uni.lassy.excalibur.examples.icrash.tutorial.v0.2
lu.uni.lassy.excalibur.examples.icrash.tutorial.v0.31
```

lu.uni.lassy.excalibur.examples.icrash.simulation

Steps	Satisfaction
001	
1->2	✓ Success
2->3	✓ Success
3->4	✓ Success
4->5	✓ Success
5->6	✓ Success
6->7	✓ Success
7->8	✓ Success
8->9	✓ Success
9->10	✓ Success
10->11	✓ Success
11->12	✓ Success
12->13	✓ Success
13->14	✓ Success
14->15	✓ Success
15->16	✓ Success

in-out	Event Name	Actor Type	oid	Interface Type	oid
trigger	oeAddCoordinator	actAdministrator	34	outactAdministrator	
sent	ieCoordinatorAdded	actAdministrator	34	inactAdministrator	

Event Parameter Name / Type	Event Parameter Value
DT dtCoordinatorID	
value : ptString	'1'
DT dtLogin	
value : ptString	steve
DT dtPassword	
value : ptString	pwdMessirExcalibur2017

Figure 25. MESSIR Requirements V&V using the **EXCALIBUR** SE Workbench.

3.11. Messir in Software Engineering Education

This section presents a knowledge transfer initiative called **MESSEP** for "**MES**sir **S**oftware **E**ngineering project courses **P**roduct line". The idea is to allow an instructor to derive a full project course, including syllabus, project description and project inputs including a software engineering environment as described in the previous section. The possible variation points are: **SWEBOK** (knowledge areas, topics, sub-topics), **Application Domains** [63] (market, categories, sub-Categories) and **IT Technologies**.

Two derivation processes are possible: forward and back-and-forth. The forward process requests to: bind the variabilities, define deltas (add/modify variation points, variants, constraints) and derive the SE project variant. The back-and-forth process requests to: select a SE project variant from the product line; define deltas (optional); derive a refined SE project variant (optional).

This allows to engineer a full SE project that can be included in any courses at bachelor or master level (using the International Standard Classification of Education ISCED [40]). At bachelor level, the main courses that could benefit from **MESSEP** are: Requirements engineering with use cases; Practical development projects with Java, JavaFx, MySQL; Introduction to software engineering concepts; Introduction to development methods concepts; Introduction to product quality; Verification and Validation. At master level, it could be: Advanced Requirements Engineering; Model Driven Engineering; Domain Specific Languages: concepts and tools; Software Engineering Environments: use and development; Formal Methods; Operational and Axiomatic Semantics; Testing and Model checking; Constraint logic programming.

Figure 26 provides a synthesis of the main project course characteristics obtained using the back-and-forth derivation process. It results in a project for bachelor ICSED level 655 that has two periods of two phases spread over a full year program, focuses on the SWEBOK knowledge areas KA1 (Requirements), KA9 (Modeling) and KA3 (Construction). Figure 27 presents the detailed coverage of the SWEBOK knowledge areas for our SE Project course variant derived. It shows that a global coverage of 36% of the SWEBOK is obtained with the course. The detailed coverage is given and, for instance, it indicates not surprisingly that 80% of the SWEBOK knowledge related to the Software Requirements knowledge area are covered by the course.

For what concerns long-life learning, professional trainings for software engineering can be set up using the **MESSIR** approach presented in this article (two experiments will be made with industrial partners in the near future). It can be a solution to increase the knowledge level on the SWEBOK knowledge areas. Since the job offers in computing and mainly in software development will increase and be 3 times higher than the degrees awarded, the SE knowledge level might decrease if no life-long learning solutions are developed. The flexible scientific approach proposed by **MESSIR** is a limited but real contribution to raising the scientific level of software engineers.

In SE research, **MESSIR** can be used to tackle an interesting pool of open research problems mainly in the following areas: Domain-Specific Languages, Model-Driven Engineering, Specification-based testing, Dependability requirements or Simulation and verification of modeling languages and, problem-driven development using constraint-oriented specifications.

Features	**Details**
Project Name	*iCrash* v 1.0
ISCED Level	BA 655 (Bachelor/Professional/First degree)
Schedule	10 hours * 14 weeks * 2 periods
Group Size	4 [2-4]
Phases	Per.1 [Pha.1/6w + Pha.2/8w] Per.2 [Pha.1/8w + Pha.2/7w]
Main SWEBOK KAs	KA1/REQ + KA9/MOD + KA3/CONS
Main Market	Applications/Collaborative Applications/Team Collaborative Applications
Main Technologies	BUILT ON eclipse, excalibur, UML Designer, Xtext, Sirius, e(fx)clipse, ORACLE VirtualBox, UML, LaTeX, PDF, Atlassian, JIRA, Confluence, Bamboo, SUBVERSION, Java, JavaFx, MySQL, gmail

Figure 26. Reference Card of a SE Project Course Variant.

Nb	Knowledge Area	Cov. (%)
1	Software Requirements	80
9	Software Engineering Models and Methods	75
7	Software Engineering Management	67
11	Software Engineering Professional Practice	63
2	Software Design	46
3	Software Construction	39
8	Software Engineering Process	33
4	Software Testing	32
5	Software Maintenance	28
14	Mathematical Foundations	19
15	Engineering Foundations	18
10	Software Quality	17
13	Computing Foundations	11
12	Software Engineering Economics	8
6	Software Configuration Management	0

Figure 27. Example of SWEBOK Coverage for SE Project Course Variant.

4. Informal Assessment of the Proposed Approach in an Education Context

In order to verify that the **MESSIR** approach really impacts positively requirements engineering, some experiments have been conducted in an academic setting at bachelor and master degrees level. During 3 years, around 80 projects have been developed in which the requirements were engineered using the **MESSIR** approach. An analysis has been made in order to determine the benefits of using the **MESSIR** approach together with the **EXCALIBUR** tool. We only present here the main facts noticed and verified during our applied experiences:

- The use case instances are efficient to ensure a complete and consistent definition of the set of system operations;
- The scientific level choice allows to invest time coherently w.r.t. to the criticality of system operations;

- The unique capability of **MESSIR** to have simulation of axiomatic requirements impacts positively the reliability of the system by avoiding to implement invalid requirements;
- Students with **EXCALIBUR** were producing the project **MESSIR** requirement documentation in a time which was twice to four times faster than students that used a tool set made of Microsoft word and UML free open source tool (e.g., http://www.umletino.com, accessed 28 February 2022);
- The average knowledge level acquired on the software engineering notions covered by the **MESSIR** approach is higher by 1 Bloom level [64] compared to a project without the **MESSIR** method.

It has also been evaluated that the **MESSEP** process can be used to set up requirements engineering courses at bachelor and master levels at two universities (University of Geneva and Peter the Great Saint-Petersburg Polytechnic University). Thanks to the derivation processes, new courses have been defined in a short period of time (2 days), allowing to set up a new course improving the existing requirements engineering courses. Thanks to the learning material available, the learning curve for the tutors has been reduced to a manageable amount which is critical in academic environment in which teachers have low time budget to re-engineer their lectures.

Since no formal data were collected, a rigorous assessment would be to define and conduct such measures as future work. To this aim, some metrics should be developed, data acquisition should be made on a significant population of requirements engineering projects. The current setting would only allow to proceed to assessment in an education context. To this aim, our work presented in [39] could be extended and adapted for by refining the SWEBOK Requirements Engineering covering in sub-topics related to the **MESSIR** methodology.

5. Conclusions and Perspectives

Software development is an activity difficult to master both technically and scientifically. This is due to many reasons, such as the rapid development rate both of the technologies available, and of the quantity of products developed and requested by the society. The challenge for the software engineering domain is to produce SE theories, methods and tools that are efficient for the industry needs, at the right cost, at the right time. Many unitary results exist but the complexity also resides in integrating those results, making them ready to use, and ensuring the necessary knowledge transfer. The **MESSIR** approach, developed during the last years and presented in this paper, aims at contributing in that direction for what concerns requirements engineering research and education. We believe that this approach integrates and improves some of the SE theories, methods and tools and offers an solution for efficient knowledge transfer. Among all the contributions brought, the main achievements we foresee are: an improved use case modeling approach, a declarative and executable requirement specification technique, a consistent integration of requirements analysis components using model-driven engineering techniques, a flexible scientific engineering of requirements analysis, and a pragmatic knowledge transfer process for software engineering education based on the standard body of knowledge for software engineering (SWEBOK).

The **EXCALIBUR** tool is limited in coverage since it handles the functional requirements part of requirements engineering and that, even if it includes independent modules that can be used for other development phases, it does not support them in an integrated and guided manner. The next version of **EXCALIBUR** should first cover non-functional requirements using the ISO/SQUARE standard and propose design models linked to requirements models.

High priority perspectives are to deploy the approach in education (a network of 12 partners is currently in development and new SE project courses are setup with the approach introduced in this paper) and industry (some professional requirement analysis training are in preparation for professional engineers). Concerning the evolution of the

MESSIR approach, it is first planned to develop a DSL for modeling non-functional requirements definition and to integrate the modeling of dependability requirements specifications. This will be done keeping the flexible scientific levels, which imply ensuring the soundness of the theories and the consistency of the methods and the tools.

For what concerns the future of **MESSIR** in industry, our experience as expert for the court of justice for conformance questions showed that there is a critical need of standardization and professionalization of the contractual collaboration. It would be an improvements to create a neutral certification third-party that would be in charge of producing the IT system requirements. The produced requirements would be the basis for the IT system provider/customer contract. The certification body will also be responsible of the system conformance evaluation necessary for the customer. To this aim, the possibility to use **MESSIR** on concrete industrial projects by creating a Requirement Engineering and Conformance Certification spin-off will be studied. This will allow **MESSIR** to also have its own requirements coming from concrete industrial needs for requirements engineering.

For a usage of **MESSIR** in small companies such as start-ups, even though Messir has been designed to be flexible, its use in a context of a startup would need some adaptations. More precisely, we would need to provide a mode presenting only the description level, to avoid the need of using latex for the introduction section by adding in the messir grammar a section for introduction, and it would be necessary to reformat the documentation using IEEE/ISO standards for requirements documentation [47].

Funding: This research received no external funding.

Acknowledgments: The author would like to thank all the persons that helped directly or indirectly to have this work exist. In particular, Alfredo Capozucca and Benoit Ries, the designers and producers of nearly all the software engineering environment components for **MESSIR**, who are "highly guilty" for the existence and the quality of **EXCALIBUR**.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Bauer, F.L. Software Engineering: 1. Foundations and Systems. In *International Federation for Information Processing: IFIP Congress Series* ; UNESDOC Digital Library: Ljubljana, Yugoslavia, 1971; pp. 530–538, ISBN 0-7204-2063-6.
2. Humphrey, W.S. *Managing the Software Process*; AddisonWesley: Boston, MA, USA , 1989.
3. ISO/IEC. *Software Engineering—Guide to the Software Engineering Body of Knowledge (SWEBOK)*; ISO-IEC TR 19759-2014; International Organization for Standardization: Geneva, Switzerland, 2014.
4. Abrial, J.R.; Hoare, A. *The B-Book: Assigning Programs to Meanings*; Cambridge University Press: Cambridge, UK, 2005.
5. Spivey, J.M. The Z Notation: A Reference Manual. In *International Series in Computer Science* ; Prentice Hall: Hoboken, NJ, USA, 1992; ISBN 978-0-13-978529-0.
6. Bjørner, D. The vienna development method (VDM). In *Mathematical Studies of Information Processing*; Springer: Berlin/Heidelberg, Germany, 1979; pp. 326–359.
7. Milner, R.; Parrow, J.; Walker, D. A calculus of mobile processes, i. *Inf. Comput.* **1992**, *100*, 1–40. [CrossRef]
8. Petri, C.A. Kommunikation Mit Automaten. Universitat Hamburg. 1962. Available online: https://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/pdf/diss_petri_d.pdf (accessed on 1 January 2022).
9. ISO/IEC. *Information Technology—Object Management Group Unified Modeling Language OMG UML—Part 1: Infrastructure*; ISO/IEC 19505-1-2012; International Organization for Standardization: Geneva, Switzerland, 2012.
10. Kruchten, P. *The Rational Unified Process: An Introduction*; Addison-Wesley Professional: Boston, MA, USA, 2004.
11. Kim, D. The State of Scrum: Benchmarks and Guidelines. Available online: https://www.scrumalliance.org/ScrumRedesignDEVSite/media/ScrumAllianceMedia/Files%20and%20PDFs/State%20of%20Scrum/2013-State-of-Scrum-Report_062713_final.pdf (accessed on 1 January 2022).
12. Beck, K. *Extreme Programming Explained: Embrace Change*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2000.
13. Bjørner, D.; Havelund, K. 40 years of formal methods. In *International Symposium on Formal Methods*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 42–61.
14. Russo, D. The Agile Success Model: A Mixed Methods Study of a Large-Scale Agile Transformation. *ACM Trans. Softw. Eng. Methodol.* **2021**, *30*, 1–46. [CrossRef]
15. Bollinger, T.; McGowan, C. A Critical Look at Software Capability Evaluations: An Update. *Softw. IEEE* **2009**, *26*, 80–83. [CrossRef]

16. Fernández, D.M.; Wagner, S.; Kalinowski, M.; Felderer, M.; Mafra, P.; Vetrò, A.; Conte, T.; Christiansson, M.T.; Greer, D.; Lassenius, C.; et al. Naming the pain in requirements engineering. *Empir. Softw. Eng.* **2017**, *22*, 2298–2338. [CrossRef]
17. Verner, J.; Cox, K.; Bleistein, S.; Cerpa, N. Requirements engineering and software project success: An industrial survey in Australia and the US. *Australas. J. Inf. Syst.* **2015**, *13*. [CrossRef]
18. Jacobson, I.; Ng, P.W.; McMahon, P.E.; Goedicke, M. *The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!* Morgan & Claypool: San Rafael, CA, USA, 2019; ISBN 9781947487277.
19. OMG. Model Driven Architecture (MDA), MDA Guide rev. 2.0 ormsc/2014-06-01, Version 1.2. 2014. Available online: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf (accessed on 1 January 2022).
20. OMG. MOF 2.0 Core Final Adopted Specification. 2014. Available online: https://www.omg.org/spec/MOF/2.0/PDF (accessed on 1 January 2022).
21. ISO/IEC. *Information Technology—Object Management Group Unified Modeling Language (OMG UML)—Part 2: Superstructure*; ISO/IEC 19507:2012; International Organization for Standardization: Boston, MA, USA, 2012.
22. Eclipse. The Eclipse Fondation. 2015. Available online: http://www.eclipse.org (accessed on 1 January 2022).
23. Xtext. Xtext—Language Development Made Easy. 2015. Available online: http://www.eclipse.org/Xtext/ (accessed on 1 January 2022).
24. Sirius. Sirius—The Easiest Way to Get Your Own Modeling Tool. 2015. Available online: http://www.eclipse.org/sirius/ (accessed on 1 January 2022).
25. SICS. SICStus Prolog—ISO Standard Compliant Prolog Development System. 2015. Available online: https://sicstus.sics.se/ (accessed on 1 January 2022).
26. SEI, C.P. *CMMI for Development (CMMI-DEV)*; Technical Report; CMU/SEI-2010-TR-033; Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, USA, 2010. [CrossRef]
27. CMMI Institute. CMMI Appraisal Results. 2015. Available online: https://sas.cmmiinstitute.com/pars/pars.aspx (accessed on 1 January 2022).
28. NDIA Systems Engineering Division. CMMI Status Report. 2015. Available online: http://www.ndia.org/Divisions/Divisions/SystemsEngineering/Documents/Past%20Meetings/Division%20Meeting%20-%20June%202011/NDIA%20SED%20June%202011CMMI%20Status%20v1.pdf (accessed on 5 July 2015).
29. Salman, R.H. Exploring Capability Maturity Models and Relevant Practices as Solutions Addressing IT Service Offshoring Project Issues. Ph.D. Thesis, Portland State University, Portland, Poland, 2014.
30. Stavru, S. A critical examination of recent industrial surveys on agile method usage. *J. Syst. Softw.* **2014**, *94*, 87–97. [CrossRef]
31. Rodríguez, P.; Markkula, J.; Oivo, M.; Turula, K. Survey on agile and LEAN usage in Finnish software industry. In Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Lund, Sweden, 20–21 September 2012; pp. 139–148.
32. VersionOne. *9th Annual State of Agile Survey*; Technical Report. 2015. Available online: https://www.watermarklearning.com/downloads/state-of-agile-development-survey.pdf (accessed on 1 January 2022).
33. VersionOne. *6th Annual State of Agile Survey*; Technical Report. 2012. Available online: https://digital.ai/resource-center/analyst-reports/state-of-agile-report (accessed on 1 January 2022).
34. Hutchinson, J.; Whittle, J.; Rouncefield, M.; Kristoffersen, S. Empirical assessment of MDE in industry. In Proceedings of the 33rd International Conference on Software Engineering, Waikiki, HI, USA, 21–28 May 2011; pp. 471–480.
35. ISO/IEC. *Software Engineering—Guide to the Software Engineering Body of Knowledge (SWEBOK)*; ISO-IEC TR 19759-2005; International Organization for Standardization: Boston, MA, USA, 2005.
36. ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013*; 2013. Available online: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf (accessed on 1 January 2022).
37. ACM/IEEE. *Software Engineering 2014—Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*; ACM: New York, NY, USA, 2015.
38. Pyster, A. *Graduate Software Engineering 2009 (GSwE2009) Curriculum Guidelines for Graduate Degree Programs in Software Engineering*; Stevens Institute of Technology: Hoboken, NJ, USA, 2009.
39. Guelfi, N.; Capozucca, A.; Ries, B. Measuring the SWEBOK Coverage: An Approach and a Tool. In Proceedings of the SWEBoK Evolution—Virtual Town Hall Meeting, Virtual Event, 25 August 2016; Virtual presentation of accepted peer reviewed paper.
40. UNESCO. *International Standard Classification of Education (ISCED) 2011*; UNESCO Institute for Statistics: Montreal, QC, Canada, 2012. [CrossRef]
41. U.S. Bureau of Labor Statistics. Employment Projections. 2015. Available online: http://www.bls.gov/emp/ep_table_102.htm (accessed on 5 July 2015).
42. Wagner, S.; Fernández, D.M.; Felderer, M.; Vetrò, A.; Kalinowski, M.; Wieringa, R.; Pfahl, D.; Conte, T.; Christiansson, M.T.; Greer, D.; et al. Status quo in requirements engineering: A theory and a global family of surveys. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2019**, *28*, 1–48. [CrossRef]
43. Fricker, S.A.; Grau, R.; Zwingli, A. Requirements engineering: Best practice. In *Requirements Engineering for Digital Health*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 25–46.
44. Burgueño, L.; Vallecillo, A.; Gogolla, M. Teaching UML and OCL models and their validation to software engineering students: An experience report. *Comput. Sci. Educ.* **2018**, *28*, 23–41. [CrossRef]

45. Gogolla, M.; Hilken, F. Model validation and verification options in a contemporary UML and OCL analysis tool. In *Modellierung 2016*; Gesellschaft für Informatik: Bonn, Germany, 2016.
46. Burgueño, L.; Izquierdo, J.L.C.; Planas, E. An empirical study on the impact of introducing a modeling tool in a Requirement Engineering course. In Proceedings of the 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Fukuoka, Japan, 10–15 October 2021; pp. 712–720.
47. ISO/IEC. *Systems and Software Engineering—Life Cycle Processes—Requirements Engineering*; ISO/IEC/IEEE 29148:2011; International Organization for Standardization: Boston, MA, USA, 2011.
48. Planas, E.; Cabot, J. How are UML class diagrams built in practice? A usability study of two UML tools: Magicdraw and Papyrus. *Comput. Stand. Interfaces* **2020**, *67*, 103363. [CrossRef]
49. Coleman, D.; Arnold, P.; Bodoff, S.; Dollin, C.; Gilchrist, H.; Hayes, F.; Jeremaes, P. *Object-Oriented Development: The FUSION Method*; Prentice-Hall, Inc.: Hoboken, NJ, USA, 1994.
50. Cotton, T. Evolutionary fusion: A customer-oriented incremental life cycle for fusion. *Hewlett Packard J.* **1996**, *47*, 25–26.
51. Guelfi, N.; Capozucca, A.; Ries, B. Website of the Messir Method and the Excalibur Environment. 2017. Available online: https://messir.uni.lu/confluence/display/MES/Messir+Requirements+Analysis+Methodology (accessed on 23 August 2017).
52. Mussbacher, G.; Al Abed, W.; Alam, O.; Ali, S.; Beugnard, A.; Bonnet, V.; Bræk, R.; Capozucca, A.; Cheng, B.H.; Fatima, U.; et al. Comparing six modeling approaches. In *Models in Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 217–243.
53. Kienzle, J.; Guelfi, N.; Mustafiz, S. Crisis management systems: A case study for aspect-oriented modeling. In *Transactions on Aspect-Oriented Software Development VII*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 1–22.
54. Guelfi, N. *Messir Tutorial*; Technical Report; University of Luxembour: Esch-sur-Alzette, Luxembourg, 2022.
55. Jacobson, I.; Christerson, M.; Jonsson, P.; Overgaard, G. *Object-Oriented Software Engineering. A Use Case Driven Approach*; c1992, Revised Printing; ACM Press: New York, NY, USA; Wokingham, UK; Addison-Wesley Pub.: Reading, MA, USA, 1992; Volume 1.
56. Cockburn, A. *Writing Effective Use Cases*; Addison-Wesley: Reading, MA, USA, 2001.
57. Armour, F.; Miller, G. *Advanced Use Case Modeling: Software Systems*; Addison-Wesley: Reading, MA, USA, 2001.
58. Jackson, M. The meaning of requirements. *Ann. Softw. Eng.* **1997**, *3*, 5–21. [CrossRef]
59. Capozucca, A.; Ries, B. Website of the Excalibur Workbench for Messir. 2017. Available online: https://messir.uni.lu/confluence/display/EXCALIBUR/Excalibur (accessed on 23 August 2017).
60. ISO/IEC. *ISO/IEC 25010—Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*; ISO/IEC 13211-1; ISO: Geneva, Switzerland, 2011.
61. Guelfi, N. A formal framework for dependability and resilience from a software engineering perspective. *Cent. Eur. J. Comput. Sci.* **2011**, *1*, 294–328. [CrossRef]
62. Ries, B.; Capozucca, A.; Guelfi, N. Messir: A text-first DSL-based approach for UML requirements engineering (tool demo). In Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, Boston, MA, USA, 5–6 November 2018; pp. 103–107.
63. Heiman, R. IDC's Software Taxonomy 2010. In *International Data Corporation*; International Data Corporation: Framingham, MA, USA, 2010.
64. Bloom, B.S.; Krathwohl, D.R. Taxonomy of educational objectives: The classification of educational goals. In *Handbook I: Cognitive Domain*; Longman Eds: New York, NY, USA, 1956; ISBN 058228239X.