The role of formalism in system requirements

JEAN-MICHEL BRUEL, University of Toulouse, IRIT
SOPHIE EBERSOLD, University of Toulouse, IRIT
FLORIAN GALINIER, University of Toulouse, IRIT
MANUEL MAZZARA, Innopolis University
ALEXANDR NAUMCHEV, Innopolis University
BERTRAND MEYER, Schaffhausen Institute of Technology and Innopolis University

A major determinant of the quality of software systems is the quality of their requirements, which should be both understandable and precise. Most requirements are written in natural language, good for understandability but lacking in precision.

To make requirements precise, researchers have for years advocated the use of mathematics-based notations and methods, known as "formal". Many exist, differing in their style, scope and applicability. The present survey discusses some of the main formal approaches and compares them to informal methods.

The analysis uses a set of 9 complementary criteria, such as level of abstraction, tool availability, traceability support. It classifies the approaches into five categories based on their principal style for specifying requirements: natural-language, semi-formal, automata/graph, mathematical, seamless (programming-language-based). It includes examples from all of these categories, altogether 21 different approaches, including for example SysML, Relax, Eiffel, Event-B, Alloy.

The review discusses a number of open questions, including seamlessness, the role of tools and education, and how to make industrial applications benefit more from the contributions of formal approaches.

ACM Reference Format:
Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, Manuel Mazzara, Alexandr Naumchev, and Bertrand Meyer. 2021. The role of formalism in system requirements. *ACM Comput. Surv.* 1, 1, Article 1 (January 2021), 35 pages. https://doi.org/10.1145/3448975

1 INTRODUCTION

In a world where software pervades every aspect of our lives, a core issue for the IT industry is how to guarantee the quality of the systems it produces. Software quality is a complex and widely studied topic, but it is not hard to provide a simple definition: quality means that *the software does the right things, and does them right.* These "things" that a software system does are known as its **requirements**. Not surprisingly, requirements engineering is a core area of software engineering.

Both goals, doing the right things and doing things right, are dependent on the quality of the requirements: the requirements must define the system so that it will satisfy user needs; and they must make it possible to assess a candidate implementation against this definition, a task

Authors' addresses: Jean-Michel Bruel, bruel@irit.fr, University of Toulouse, IRIT; Sophie Ebersold, sophie.ebersold@irit.fr, University of Toulouse, IRIT; Florian Galinier, florian.galinier@irit.fr, University of Toulouse, IRIT; Manuel Mazzara, m.mazzara@innopolis.ru, Innopolis University; Alexandr Naumchev, a.naumchev@innopolis.ru, Innopolis University; Bertrand Meyer, Bertrand.Meyer@inf.ethz.ch, Schaffhausen Institute of Technology and Innopolis University.

known as *validation* (as distinct from *verification*, which assesses the internal properties of the implementation).

Validation can only be effective if the requirements are precise. Precision in science and technology is typically achieved by using mathematical methods and notations, also known in software engineering as *formal* methods and notations. This survey examines the state of the art in applying such formal approaches to software requirements, and perspectives for their further development.

Precision is so important that an outsider might assume that *all* software development starts with a formal description of the requirements. This approach is by far not the standard today; most projects, if they do write requirements, express them informally, either in a natural-language "requirements document" or (in agile methods) through individual "user stories", also expressed in natural language.

Most approaches, particularly formal ones, focus on requirements per se, not directly connected to design and implementation. We will also encounter *seamless* approaches which, in contrast, seek to unify the software process by making requirements benefit from concepts, notations and tools that are also applicable to these other tasks.

For length reasons, a few sections with special numbering such as 4.1.A, as well as two appendices, numbered A and B, appear only in the addendum to this article, available online [16].

1.1 Terminology: requirements versus specifications

Previous surveys [40, 41, 48, 67, 121, 133] have covered formal *specification*. The present one is about formality in *requirements*. The two concepts are close: both specifications and requirements describe the "what" of a system or system element rather than the "how". The literature accepts that there is a difference or at least a nuance, although various interpretations of that difference remain (even within a single normative source such as the Software Engineering Body of Knowledge [14]). The general idea is that requirements are closer to the user view and specification to the system view:

- Requirements describe the properties of a system part that are relevant to its *users and environment*; for example, what functionality a Web browser provides to its users and the constraints on this functionality.
- A specification describes the *technical properties* of some system part; for example, how the rendering engine of the browser must display any given HTML text.

This definition shows why the distinction cannot be absolute: to produce the requirements of a complex system, one must decompose them repeatedly into sub-requirements of its components. Each iteration of this process becomes more detailed, more technical and hence closer to a specification.

Another distinction is sometimes encountered in discussions of requirements: between "user" and "system" requirements. Defining the distinction precisely is elusive, and it does not seem to bring significant value to discussions of requirements. Correspondingly, this article does not rely on it.

1.2 Terminology: "specification"

Two further observations on terminology will help avoid confusion:

- Aside from its technical software engineering meaning discussed above, "specify" is used in ordinary English to mean "describe", or "mention", or just "state". Such usage is applicable to requirements, as in "the requirements document specifies that this case is an error". It occurs in the present discussion when there is no risk of confusion with the technical meaning.

- Following the standard practice of the requirements literature, this article uses “a *requirement*” as the description of a particular property of a system and “the *requirements*” as the collection of every such individual “requirement” for a system. “Requirements” here is not just the plural of “requirement” but a concept on its own, often understood as an abbreviation for “the requirements document”. In this discussion, “requirements” denotes that collective concept. When the emphasis is on one or more specific “requirement” the phrasing will reflect it clearly, as in “requirement #25” or “the following three requirement *elements*”.

2 RUNNING EXAMPLE

Publications describing a requirements approach generally use specific examples, chosen to highlight its advantages. The present survey instead relies on one example which it expresses in every approach. An objection to this solution is that the example may be more suited to some approaches than to others. It is offset by the obvious benefits: making the discussion easier to read (by requiring the reader to learn only one example) and enabling meaningful comparisons.

An earlier single-example comparison was Wing’s 1988 study of specifications of a library system [132], but it is too simple to reflect the challenges of today’s demanding IT applications. The example of this survey is the Landing Gear System (LGS) for airplanes, a case study [13] that has received wide attention [7, 89, 117]. LGS is a complex, critical system for which the requirements involve diverse stakeholders and many fields of expertise.

Physically, an LGS consists of the landing set, a gear box to store the gear when retracted, and a door attached to the box. The door and gear are independently activated by a digital controller. The controller reacts to a handle’s changes of position by initiating gear extension or retraction. It must align in time the events of changing the handle’s position and sending commands to the door and the gear actuators: doors can be opened and closed and the gear is moving either out (extension) or in (retraction). The following rules, precise although still in natural language, express these properties:

- (R11bis)[1] If the landing gear command handle has been pushed down and stays down, then eventually the gear will be locked down and the doors will be seen closed.
- (R12bis) If the landing gear command handle has been pushed up and stays up, then eventually the gear will be locked retracted and the doors will be seen closed.
- (R21) If the landing gear command handle remains in the down position, then retraction sequence is not observed.
- (R22) If the landing gear command handle remains in the up position, then extension sequence is not observed.

We will see how to express such properties in some of the approaches surveyed.

3 CLASSIFYING APPROACHES TO REQUIREMENTS

This section introduces a classification of approaches into five categories (section 3.1), presents the approaches retained and why they were retained (section 3.2), and lists the assessment criteria (section 3.3).

3.1 The classification

Approaches fall into five categories based on their primary way of representing requirements:

- *Natural language* approaches primarily express requirements in English or another human language, although (section 4.1) they can restrict expressiveness to a subset of that language.

[1]The R11 and R12 requirements from [13] are temporal, R11bis and R12bis are the non-temporal alternative.

- *Semi-formal* approaches codify the form of requirements, in effect defining a precise requirements language, which is neither a mathematical notation (as in the next two categories) nor derived from a programming language (as in the last category).
- *Automata/graphs* approaches rely on automata or graph theory. They often come with graphical support, which may allow using them without mastering the underlying mathematics.
- *Mathematical* approaches rely on mathematical formalisms other than those of the previous category:. The theoretical basis is generally set theory or universal algebra.
- *Seamless, programming-language-based* approaches integrate requirements closely with other software tasks (such as design and implementation), using a programming language as notation.

3.2 Selection criteria and list of approaches surveyed

Table 1 lists the retained approaches, each with references to the associated publications in the bibliography. When these publications do not give the approach an explicit name, we devised one reflecting the central concept, for example "Requirements Grammar", and marked it with an asterisk*. Due to space limitations, the discussion of some approaches appears only in an addendum to the paper available online [16]; the corresponding section numbers contain a letter (e.g. 4.1.A).

Name	Category	References	Section
Requirements Grammar*	Natural language	[111]	4.1.1
Relax		[130]	4.1.2
Stimulus		[59]	4.1.3
NL to OWL*		[65]	4.1.4
NL to OCL*		[45]	4.1.A (addendum)
NL to STD*		[4]	4.1.B (addendum)
Reqtify	Semi-formal	[118]	4.2.1
KAOS		[127]	4.2.2
SysML		[94]	4.2.3
URN		[6]	4.2.A (addendum)
URML		[11]	4.2.B (addendum)
Statecharts	Automata- or graph-based	[47]	4.3.1
Problem Frames		[58]	4.3.2
FSP/LTSA		[69]	4.3.3
Petri Nets		[99]	4.3.A (addendum)
Event-B	Mathematical notation	[1]	4.4.1
Alloy		[57]	4.4.2
FORM-L		[91]	4.4.3
VDM		[12]	4.4.A (addendum)
Tabular Relations		[98]	4.4.B (addendum)
Multirequirements	Seamless, PL based	[75]	4.5.1
SOOR		[87]	4.5.2

Table 1. Requirements approaches surveyed.

Requirements engineering is rich with methods and tools. Any survey must involve a choice and, inevitably, some subjectivity. The approaches retained meet one or more of the following criteria:

- Widely used (as in the case of commercially available tools such as Reqtify).
- Widely publicized.
- Influential.
- Possessing, in the authors' view, other distinctive characteristics that warrant discussion.

Note that some of the authors have (separately) been involved in three of the methods reviewed: Relax (4.1.2), multirequirements (4.5.1), and seamless requirements (4.5.2).

Among the inevitable omissions are historical contributions whose concepts reappear in some of the included approaches, particularly Z [2] and Hoare logic [50], which influenced Event-B (4.4.1), Alloy (4.4.2) and Design by Contract (4.5). Some major techniques of specification and verification that do not fall into the category of requirements approaches include the TLA+ method and tool [61] and many developments around the SPIN/Promela model-checking environment [113] .

3.3 Criteria for assessing approaches

The matter of assessing the quality of *requirements* has received significant attention, not only in the research literature [29, 109] but also in industry standards, from the venerable IEEE 830-1993 [53] to the more recent ISO/IEC/IEEE 29148-2011 [54]. They list such criteria as traceability, verifiability, consistency, justifiability and completeness. For the present discussion, we need to assess (one notch higher in abstraction) the quality of requirements *approaches*. The assessment uses nine criteria.

Criterion 1, *System vs. Environment* (also called "*Scope*" for short), refers to the classic Jackson-Zave distinction [137] between two complementary parts of requirements: the environment (or "domain") in which the system operates, including constraints it imposes, such as "no car will travel faster than 250 km/hour" or "any bank transfer above EUR 10,000 must be reported" ; and the system (or "machine") which the project will build. Does the approach cover both, or only the system part?

Criterion 2, *Audience* (also appearing in the assessment sections as "*Prerequisites*"), addresses the level of expertise expected of people who will use the requirements. Do they need, for example, to be trained in formal methods? Or are the requirements suitable for any stakeholder?

Criterion 3, *Level of Abstraction* (abbreviated as "*Abstraction*"), addresses the level of detail (of properties of the system under description) which the requirements may or must cover.

Criterion 4, *Associated method* ("*Method*"), assesses whether the approach includes a comprehensive methodology to guide the requirements process — as opposed to method-neutral approaches, which provide requirements support but adapt to their users' preferred methods.

Criterion 5, *Traceability support* ("*Traceability*"), assesses how the approach handles one of the most important issues associated with requirements: keeping track of one- or two-way relations between requirement elements and their counterparts in design, code and other project artifacts. The IEEE standards emphasize the role of traceability as one of the key factors of requirements quality.

Criterion 6, *Non-functional requirements support* ("*Coverage*"), addresses whether the approach covers only the description of functional properties of the system (functions and environment constraints) or extends to non-functional properties such as performance and security.

Criterion 7, *Semantic definition* ("*Semantics*"), assesses whether there exists a precise (if possible, formal) definition of the approach.

Criterion 8, *Tool Support* ("*Tools*"), covers the availability of tools to support the approach.

Criterion 9, *Verifiability*, assesses whether an approach supports the possibility of formally verifying properties of the requirements. For approaches that provide such facilities, "formal verification" (taken also to include *validation*, see section 1) typically means mathematical proofs, preferably supported by tools since manual verification is not sufficient for large and complex systems.

3.4 Why these criteria?

Since the listed criteria play an important role in the review, it is legitimate to ask what justifies their choice. No such list can contend to be the only possible one. Setting aside, however, any unconscious role the authors' (diverse) experience may have played, the list results from an empirical analysis of the literature and discussions with requirements engineers. Since part of the audience for this article consists of managers and engineers in search of suitable requirements approaches, the focus is on requirements that will help them. Particularly important are properties of requirements approaches that, if misunderstood at the outset, could lead requirements authors to make the wrong choice and the project to bear the consequences. Helping the reader avoid such mistakes has been the primary guide for choosing the criteria. We may illustrate this general rule for the first few criteria:

- **Audience**: a requirements method may be excellent on its own but not adapted to the people who will use it. For example, it may require the use of technical software concepts whereas the principal stakeholders are non-software professionals who will not understand them. Conversely, it may be not technical enough for an audience of savvy software professionals (for example in a mission-critical embedded system) who will resent what they see as informality and vagueness. Either of these mistakes can jeopardize the requirements and the project. To avoid such "casting errors", it is important to state each approach's intended audience.
- **Abstraction**. Different contexts require different levels of abstraction. Some approaches are more abstract, others closer to implementation concepts. Here too a misunderstanding can lead to a wrong choice with damaging consequences for a project.
- **Method**. Another key question, critical to any project's choice, is whether an approach just provides a notation and possibly tools, without prescribing any particular requirements method, or imposes a specific process. Some projects will benefit from a comprehensive method that guides them throughout; others want to preserve their freedom to retain any method that the organization already uses for its developments.

These observations only cover three criteria, but similar justifications support the other six (tools, traceability, coverage, scope, verification, semantics). Each criterion influences the choice of approach so fundamentally, and mistakes can cause such damage, that it is essential to state how each method fares with respect to each of them. Hence the systematic assessments along these nine dimensions.

4 REVIEW OF SELECTED APPROACHES

We now explore the approaches in the order of the five categories of the previous section, illustrating them through the example introduced in section 2 and evaluating them by the criteria of section 3.3.

4.1 Natural language

Natural language is, as noted, the dominant form of requirements for projects in industry [60, 78]. A number of requirements approaches consequently start from natural language statements of requirements (sometimes processing them automatically [3, 33]). Zhao et al. report on the state of the art in this field in their mapping study [138]. Using natural language faces a fundamental challenge: software construction needs a high degree of precision, but natural language is notoriously imprecise. ([72] provides a detailed analysis of the problems of using natural language for requirements.)

Significant effort has been devoted to processing natural-language requirements automatically, with the purpose of detecting inconsistencies and, more generally, improving quality. Examples of such work include [77], from 1996, and, more recently, [9, 112].

Since natural-language processing raises challenges at the frontier of artificial intelligence research, some approaches bypass the difficulty by using a *constrained* form of natural language, ensuring some degree of precision without going as far as the semi-formal and formal approaches studied next. Examples include Requirements Grammar (section 4.1.1), Relax (4.1.2) and Stimulus (4.1.3).

Other approaches go the full way of Natural Language Processing (NLP) to extract precise information and in particular to detect inconsistencies. They include Natural Language to Web Ontology Language (NL to OWL, section 4.1.4), Natural Language to Object Constraint Language (NL to OCL, 4.1.A) and Natural Language to State Transition Diagram (NL to STD, 4.1.B).

4.1.1 Requirements Grammar. Requirements Grammar approaches define a structured subset of natural language. This is the case for EARS [71] and for the simpler approach of Scott and Cook [111], which defines a context-free requirements language. Requirements elicitation is expected to produce requirements in this language, avoiding inconsistencies. As with a programming language, the overall structure involves fixed keywords, borrowed here from English, such as **if** and **shall**, but they can be combined with free-form elements with no predefined meaning, as in

if the gear is locked down, the doors **shall** be closed

Fig. 1 shows a representation of requirement R11bis (*If the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gear will be locked down and the doors will be seen closed.*) in this approach. The boxes show the hierarchical structure.

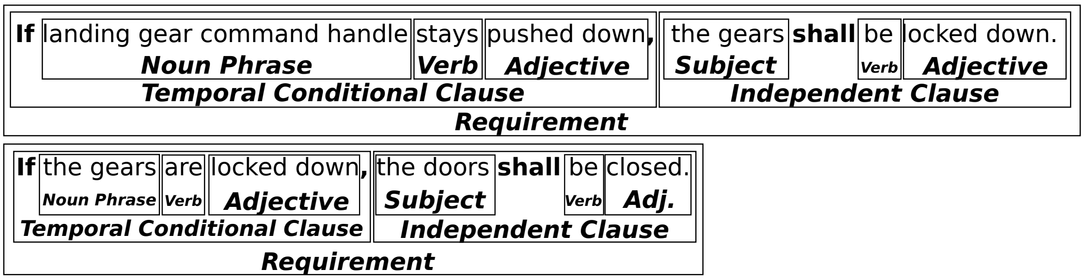


Fig. 1. Requirement R11bis in Scott and Cook's Requirements Grammar

The grammar is not able to express this requirement as stated in section 2: each requirement can only involve one *Independent Clause*, with a single *Subject of interest*. In the given requirement, there are two subjects (the *gear* and the *doors*). A solution is to split this requirement into two:

- If the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gear will be locked down.
- If the gear is locked down, then eventually the doors will be seen closed.

The structure of the specification is the following, with keywords in boldface:

Requirement: a sentence corresponding to a requirement.
  Independent Clause: the mandatory part of the requirement, describing the need.
    Subject: the subject of interest of the requirement.
    shall: a keyword.
    Verb: the action of the requirement.

Adjective or Noun Phrase: depending of the verb, a complement for the verb.
Temporal Conditional Clause: an optional constraint on the requirement.
If: a keyword.
Noun Phrase: the subject of the constraint.
Verb, adjective: as above.

Assessing Requirements Grammar according to the criteria of section 3.3:

- *Scope*: the approach can cover both system and environment aspects.
- *Audience*: since the notation, while constrained, uses a subset of natural language, requirements are readable by any stakeholder, including those who are not aware of the constraints and will only see a natural-language description, possibly looking a bit contrived.
- *Abstraction*: this approach is for requirements only, not influenced by implementation concerns.
- *Method*: the approach does not imply a particular requirements engineering method.
- *Traceability*: the approach focuses on requirements, independently of other steps and products (design, implementation), so it offers no specific support for traceability.
- *Coverage*: given that the approach expresses requirements in a very abstract form, it can include both functional and non-functional requirements.
- *Semantics*: the syntax of specifications is defined precisely (through the concept of context-free Requirements Grammar) but there is no corresponding rigorous definition of the semantics.
- *Tools*: the authors proposed a tool, Badger (which seems no longer to be available), to express requirements and analyze their lexical clauses.
- *Verifiability* is limited to consistency (the lexical clauses are analyzed to detect if all requirements follow the pattern).

4.1.2 Relax. Relax [130] is a language for formal modeling of requirements for Complex Adaptive Systems (CAS). The Relax syntax is close to natural language. Fig. 2 shows requirements R11bis, R12bis, R21 and R22 expressed in it.

R11bis: The gear SHALL be locked down and the doors SHALL be closed AS EARLY AS POSSIBLE AFTER the landing gear command handle has been pushed down.
R12bis: The gear SHALL be locked retracted and the doors SHALL be closed AS EARLY AS POSSIBLE AFTER the landing gear command handle has been pushed up.
R21: The retraction sequence SHALL not be observed AS EARLY AS POSSIBLE AFTER the command handle remains in down position.
R22: The extension sequence SHALL not be observed AS EARLY AS POSSIBLE AFTER the command handle remains in up position.

Fig. 2. Representation of Landing Gear System requirements expressed with Relax

Keywords such as `AS EARLY AS POSSIBLE` or `AFTER` express temporality of events. They are semantically defined through FBTL (fuzzy branching temporal logic [83]), making it possible to submit the requirements to validation tools. For example, a more formal representation of R11bis is:

R11bis: SHALL(AFTER('the landing gear command handle has been pushed up') AS EARLY AS POSSIBLE ('the gear is locked up and the doors is closed'))

and is translated into FBTL, per the translation rules in [130], as:

$$AG(A\mathcal{X}_{>(landing\_gear\_command=down)_d}(A\mathcal{X}_{\geq d_1}(gear = locked\_down \wedge doors = closed)))$$

expressing that in any state after the event ‘landing gear command has been pushed down’ (where $(landing\_gear\_command = down)_d$ is the time since that event), ‘gear is locked down and doors are closed’ becomes true in a near-future state, after a fuzzy duration d_1.

Relax expresses environment properties directly through the keyword ENV, defines “monitors” through MON, and states relationships between them through REL.

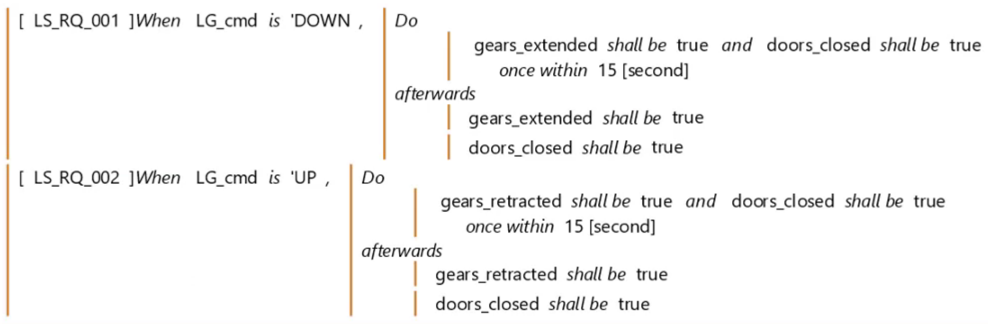
An original feature of Relax, explaining the name, is the ability to mark some requirements as critical, and to *relax* a non-critical requirement if necessary to preserve critical ones.

Assessing Relax according to the criteria of section 3.3:

- *Scope*: the approach explicitly covers system and environment aspects.
- *Audience*: the notation uses natural language expressions, making readable by any stakeholder.
- *Abstraction*: this approach is for requirements only, not influenced by implementation concerns.
- *Method*: the approach does not assume a particular requirements engineering method.
- *Traceability*: Relax focuses on requirements, independently of other steps and products (design, implementation), so it offers no specific support for traceability. Nevertheless, the language provides a way to express relationships between requirements (through the keyword DEP).
- *Coverage*: targeting adaptive systems, Relax mainly addresses functional requirements.
- *Semantics*: on the basis of a precisely defined syntax for specifications, the semantics is defined through FBTL (fuzzy branching temporal logic).
- *Tools*: only prototypes such as the Xtext editor [32] have been developed around Relax.
- *Verifiability* is linked to FBTL capabilities.

4.1.3 Stimulus. The Argosim Stimulus [115] tool expresses requirements in a natural-language-like syntax [59], similar to Relax. It is directed at stakeholders involved in system development.

Fig. 3 shows an initial attempt at expressing requirements R11bis and R12bis. It applies to a more complete version of the LGS example, taking into account timing properties from the original LGS paper not included above: a 15-second duration for retraction and for the extension sequence.



```
[ LS_RQ_001 ]When  LG_cmd  is  'DOWN ,   Do
                                              gears_extended shall be true  and  doors_closed shall be true
                                                  once within 15 [second]
                                         afterwards
                                              gears_extended shall be true
                                              doors_closed shall be true
[ LS_RQ_002 ]When  LG_cmd  is  'UP ,   Do
                                            gears_retracted shall be true  and  doors_closed shall be true
                                                once within 15 [second]
                                       afterwards
                                            gears_retracted shall be true
                                            doors_closed shall be true
```

Fig. 3. A possible expression of requirements R11bis and R12bis in Stimulus

The idea is that after writing such an initial version one may, by simulating inputs and observing outputs, detect problems and improve the description. For example, Stimulus defines the semantic of *When* as “*at the time the condition holds*”. Then R11bis as defined in Fig. 3 (*LS_RQ_001*) uses both a *When* and a *Do ... afterwards*. This means that after the handle has been pushed down, within 15 seconds the doors shall be closed and the gear down, remaining so until a new event. Without the *Do ... afterwards* clause, the *When* would mean that when the handle is down, within 15 seconds the doors shall be closed and the gear down, leaving the behavior undefined afterwards. Such wrong behavior can be detected through the Stimulus model-checker, which simulates the system behavior by varying inputs. Users can observe the system's reaction and correct the requirements as needed. Stimulus favors such a process of incremental improvement of the requirements.

Assessing Stimulus according to the criteria of section 3.3:

- *Scope*: the approach explicitly covers both system and environment aspects.
- *Audience*: the language is close to natural language, making it accessible to any stakeholder.
- *Abstraction*: this approach is for requirements only, not influenced by implementation concerns.
- *Method*: Stimulus does not assume a particular requirements engineering method.
- *Traceability*: the approach focuses on requirements, independently of other steps and products (design, implementation), so it offers no specific support for traceability.
- *Coverage*: the approach covers only functional requirements.
- *Semantics*: the language is inspired by Lucid Synchrone [23] and Lutin [104], defined in the literature with precise semantics.
- *Tools*: Stimulus is the name of both the language and the tool supporting it.
- *Verifiability* can use model-checking to simulate a system's reaction to various inputs.

4.1.4 NL to OWL. The approach of [65] translates natural-language requirements into an intermediate requirements modeling language that can be easily formalized in OWL [10].

Requirement R11bis of the LGS (*If the landing gear command handle has been pushed down and stays down, then eventually the gear will be locked down and the doors will be seen closed*) yields two functional goals:

FG11-1 := lock_down <object: {the gear}> :< <trigger: push_down <object: {landing gear command handle}> >
FG11-2 := close <object: {the doors}> :< <trigger: FG11-1> >

FG11-1 states that the gear should be locked down for a while after the landing gear command handle has been pushed down. FG11-2 states the obligation to close the doors when the gear is locked down, as triggered by the first functional goal. R12bis can be modeled in a similar way.

R21 (*When the command line is working, if the landing gear command handle remains in the down position, then retraction sequence is not observed*) can be modeled as only one functional goal:

FG5 := extend <object: {the gear}> :< <trigger: remains_down <object: {landing gear command handle}> >

This functional goal models the need to observe no retraction sequence — and hence extend the landing gear — when the handle remains down. R22 can be modeled in a similar way.

Assessing NL to OWL according to the criteria of section 3.3:

- *Scope*: NL to OWL covers both system and (through *domain assumptions*) environment.
- *Audience*: NL to OWL is for requirements engineers.
- *Abstraction*: not influenced by implementation, the approach is for requirements only.
- *Method*: NL to OWL guides the process of formalizing natural language requirements.
- *Traceability*: requirements are decomposed into ontologies, sharing the same namespace, which can be used to create links between several requirements.
- *Coverage*: both functional (*functional goals*) and non-functional (*quality goals*) properties .
- *Semantics*: the semantics of the approach comes from OWL.
- *Tools*: no tool directly supports the method; there are tools for OWL such as Protégé [102].
- *Verifiability*: NL to OWL provides no support for requirements verification (though OWL has a formal semantic definition).

Discussions of two other natural-language-based approaches, NL to OCL [45] and NL to STD [4] appear in the addendum ([16], sections 4.1.A and 4.1.B).

4.2 Semi-formal

A number of approaches, including both research efforts and industrial products, use partially formalized notations. Some of the most important are Reqtify (section 4.2.1), KAOS (4.2.2) and SysML (section 4.2.3), as well as User Requirements Notation (URN) and Unified Requirements Modeling Language (URML), both presented in the addendum (4.2.A and 4.2.B).

4.2.1 Reqtify. Reqtify [118], from Dassault Systems, is widely used in industry. It is semi-formal in the sense that it requires a partially structured approach to requirements management. Reqtify is often used jointly with Doors [51], even though Doors comes from a different provider, IBM Rational. In both cases the focus is not on producing requirements but on managing them, independently of how they were produced.

Doors is a collaborative tool allowing different stakeholders to work on requirements, typically maintained as spreadsheets, and set priorities according to levels of risks. Reqtify's focus is on traceability: the tool supports defining relationships between requirements typically expressed in natural language and coming from such tools as Microsoft Word, spreadsheets, or modeling tools.

Since these approaches do not define any specific method or notation for expressing requirements, we cannot demonstrate them on the running case study. In operational practice the requirements would be expressed in some document, e.g. Word or PDF. Reqtify would support defining and managing traceability between their various elements.

Assessing Reqtify according to the criteria of section 3.3:

- *Scope*: Reqtify can be used to specify both system and environment aspects.
- *Audience*: Reqtify is aimed at a large audience of stakeholders, without particular technical prerequisites.
- *Abstraction*: Reqtify focuses on requirements, without any influence from implementation.
- *Method*: no particular method is attached to the approach.
- *Traceability* is the strong point of Reqtify, which offers support for tracing requirements from specification to design and code. For example, Reqtify makes it possible to import requirements expressed in a Microsoft Word document and link them to C code.
- *Coverage*: Reqtify can be used for specifications of functional and non-functional requirements.
- *Semantics*: no formal semantics is associated with the approach.
- *Tools*: Reqtify *is* a software tool.
- *Verifiability*: Reqtify provides no verification methodology.

4.2.2 KAOS. KAOS [27], like i* [134], is based on the Goal-Oriented Requirements Engineering approach to requirements [127, 128, 135]. The key idea is to base requirements on a higher-level concept, goals. A goal is statement of intent expressed in terms of business needs (such as "turn more sales inquiries into actual sales" for a customer management system). Requirements then express system properties helping to achieve these goals with the help of assumptions and domain properties. Goals can be composite, expressed in terms of simpler goals through tree operators including AND, OR and "+" (denoting a less formal relation, "contributes to"). The general approach is refinement-based: start from high-level goals and decompose them using the operators. A non-composite goal is called a "requisite". The OR operator makes it possible to include alternative paths.

KAOS uses natural language to express goals and a semi-formal notation for relationships between goals, with concepts such as "milestone" and "conflict", and supports the refinement process.

Goals cover both system and environment properties: if a requisite can be assigned to an agent of the system, it is an "operational goal", describing a system property. Otherwise it is an "expectation", describing an environment property.

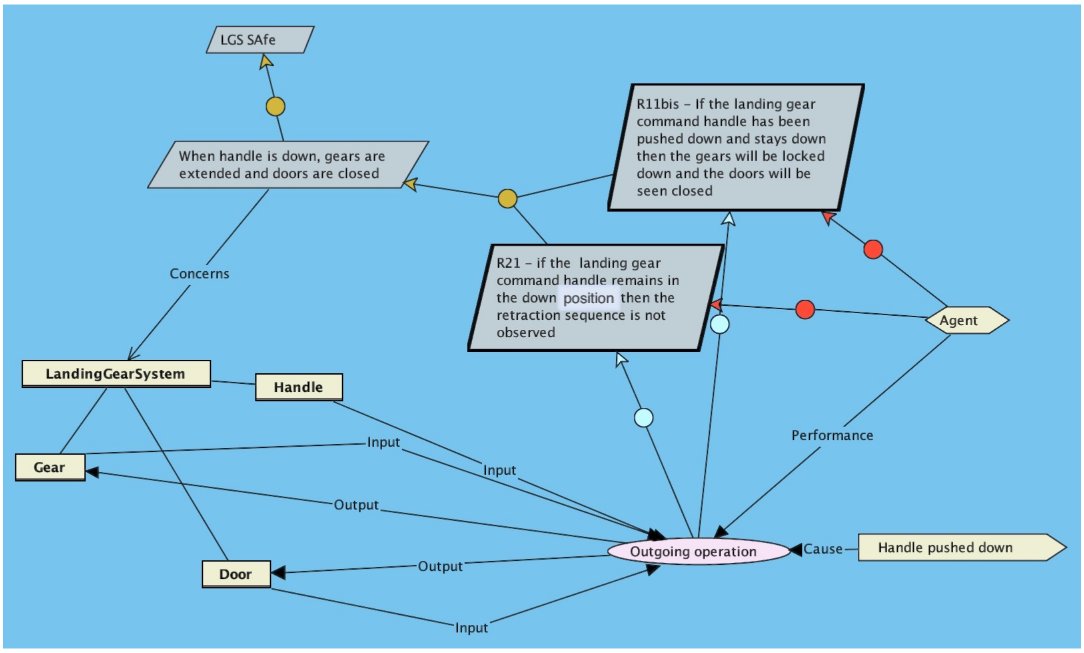

LGS SAfe
When handle is down, gears are extended and doors are closed
R11bis – If the landing gear command handle has been pushed down and stays down then the gears will be locked down and the doors will be seen closed
R21 – if the landing gear command handle remains in the down position then the retraction sequence is not observed
Concerns
Agent
LandingGearSystem
Handle
Gear
Door
Input
Input
Output
Output
Input
Performance
Outgoing operation
Cause
Handle pushed down

Fig. 4. Excerpt of the KAOS diagram of requirements R11bis and R21 of the LGS (*Objectiver*)

The KAOS model of Fig. 4 covers LGS entities including door, gear and handle. Both R11bis and R21 refine the goal ' '*When the handle is down, gear is extended and doors are closed*", which is itself part of the refinement of a more global goal defining the whole system's safety. "Outgoing operation" addresses R11bis and R21 by managing the LGS extension sequence: after the handle has been moved up, doors remain closed and the gear locked down. Some agent, triggered by the event "handle pushed down", will be responsible for this operation.

Assessing KAOS according to the criteria of section 3.3:

- *Scope*: KAOS addresses both system and environment aspects in the refinement process, through the notions of operational goal and expectation.
- *Audience*: KAOS requires modeling experience and some training in the method.
- *Abstraction*: there is no influence from implementation in KAOS. The approach focuses on a concept, goals, which is at an even higher level of abstraction than requirements.
- *Method*: KAOS includes a general methodology for modeling systems, specifying dependencies between requirements, and refining goals.
- *Traceability*: KAOS includes support for linking to specification documents.
- *Coverage*: the approach mostly handles functional requirements but can also include some non-functional ones.
- *Semantics*: in KAOS, behavioral goals can be described in temporal logic or in Event-B [1, 70].

- *Tools*: Objectiver [106] supports the expression and refinement of user requirements in KAOS.
- *Verifiability*: KAOS/Objectiver has been integrated with model-checking tools (using e.g. LTL).

4.2.3 SysML. SYStem Modelling Language [94] is an extension of UML [95] dedicated to systems engineering. SysML provides requirements diagrams, which allow users to express requirements in a textual representation and cover non-functional requirements.

SysML diagrams can express traceability links between different requirements (containment, derive, copy, trace) or between requirements and implementation elements (satisfy, verify, refine) as well as other modeling artifacts (blocks, use cases, activities). As an illustration, Fig. 5 shows the links between the LGS requirements R11bis and R21, elements that satisfy them and the landing gear outgoing sequence requirement, viewed as their parent requirement.

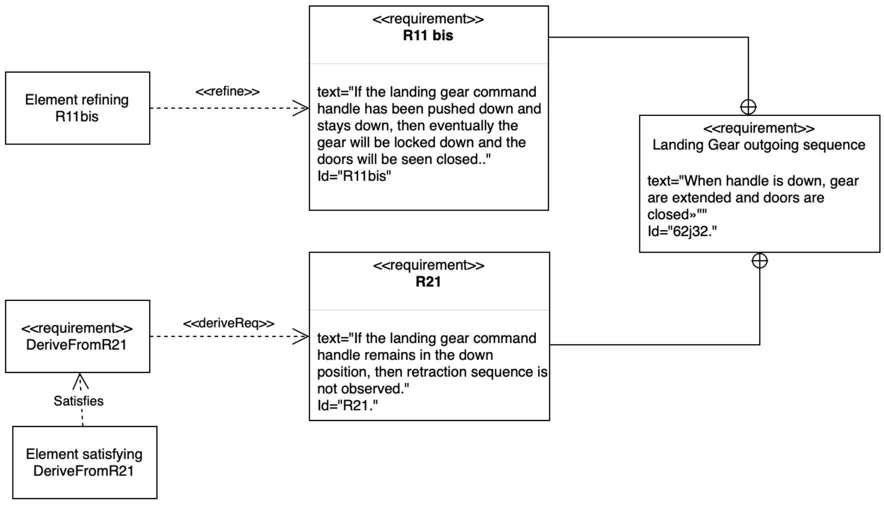

Element refining R11bis
<<refine>>
<<requirement>> R11 bis
text="If the landing gear command handle has been pushed down and stays down, then eventually the gear will be locked down and the doors will be seen closed.."
Id="R11bis"
<<requirement>> Landing Gear outgoing sequence
text="When handle is down, gear are extended and doors are closed»""
Id="62j32."
<<requirement>> DeriveFromR21
<<deriveReq>>
<<requirement>> R21
text="If the landing gear command handle remains in the down position, then retraction sequence is not observed."
Id="R21."
Satisfies
Element satisfying DeriveFromR21

Fig. 5. Excerpt of the SysML functional requirements diagram of LGS

Assessing SysML according to the criteria of section 3.3:

- *Scope*: SysML focuses on system requirements, particularly for complex systems.
- *Audience*: SysML is a modeling language needing specific knowledge.
- *Abstraction*: the approach is at a high level of abstraction. Its requirement diagrams enable quick requirements analysis and visual design.
- *Method*: SysML, like UML, is a notation, providing no methodology.
- *Traceability*: the approach provides traceability links.
- *Coverage*: SysML covers both functional and non-functional requirements.
- *Semantics*: there is not precise semantic definition of the approach.
- *Tools*: SysML is supported by a number of tools such as IBM Rhapsody [52], Modelio [82], Enterprise Architect [119] and Papyrus [35].
- *Verifiability*: only basic structural verification is possible, based on the links that have been specified (e.g. check that each requirement is supported by at least one modeling element).

4.3 Graphs and automata

A number of approaches rely on the mathematical theories of graphs and automata, well known in computer science [80] and supported by convenient graphical representations.

A common application is the modeling of dynamic aspects of a system, particularly behavior and timing. We examine Statecharts (section 4.3.1), Problem Frames (4.3.2), Finite State Processes / Labelled Transition System Analyser (FSP/LTSA, 4.3.3), and Petri nets (4.3.A, in the addendum [16]). Other approaches offering different flavors of the same concepts include UML Activity Diagrams and State Diagrams [92], UPPAAL [100], DEVS [20] and SCXML [114].

4.3.1 Finite automata, state diagrams and statecharts. The most widely used kind of automaton, notable for its simplicity and power, is the *finite* automaton, used (in applications to system modeling) through the closely related *finite-state diagram.* Such a diagram is a mathematical device defined by a finite set of *states*, each representing a possible configuration of a system or computation, the designation of some of the states as *initial* and some as *final*, and a finite set of *transitions* between states. Each transition models the effect of a given event in a certain state, by defining the resulting state or states. To model an entire system execution, the automaton starts in an initial state then processes events by following the corresponding transitions, until it reaches a final state.

Part of the attraction of finite-state diagrams is that they enjoy a natural representation as graphs, with nodes as states and transitions as edges, as illustrated in Fig. 6 which uses the most popular variant: the Statechart or " Harel chart" [47], particularly suitable for modeling parallel systems.

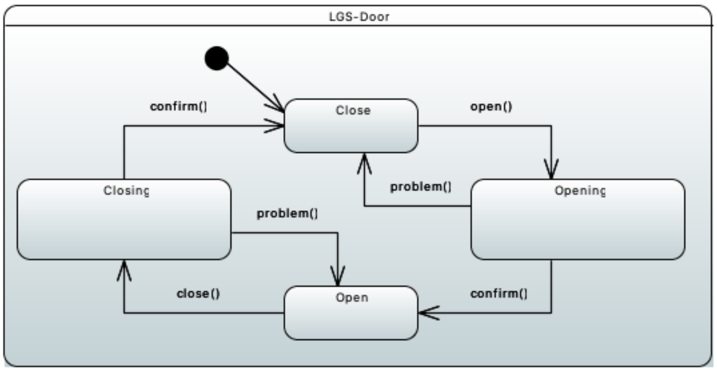


Fig. 6. A finite-state diagram for the LGS Door

Assessing finite-state diagrams, particularly Statecharts, according to the criteria of section 3.3:

- *Scope*: there is no specific modeling of the environment, although events leading to transitions can come from the outside or the inside of the system.
- *Audience*: the notation, while graphical, assumes an understanding of the semantics.
- *Abstraction*: Statecharts are meant for specification, independently of implementation concerns, and are abstract due to the specific notation. Note that available tools provide an execution environment to animate state machines (for simulation rather than actual implementation).
- *Method*: the approach enforces a strong methodological discipline, based on modeling systems in the form of states and transitions.
- *Traceability*: the approach focuses on the expression of dynamic behavior requirements and offers no specific support for traceability.
- *Coverage*: Statecharts mainly address functional requirements.
- *Semantics*: the basic semantics of Statecharts comes from the theory of finite-state automata.

- *Tools*: many tools (e.g. Stateflow) rely on Statecharts principles, sometimes redefining some of its semantics.
- *Verifiability* is supported by the semantics implementation of the tools, using model-checking.

4.3.2 Problem Frames. Problem Frames is an approach to software requirements analysis developed by Michael Jackson [58] in the nineties. It has influenced a number of subsequent approaches by bringing to light the need to divide requirements into *system* and *environment* properties.

As defined by Jackson, a *problem frame* "*defines the shape of a problem by capturing the characteristics and interconnections of the parts of the world it is concerned with, and the concerns and difficulties that are likely to arise.*" The methodology supports decomposing requirements, treated as relationships between the system and the real world. Fig. 7 shows a problem diagram with a concrete *machine domain* (`Computing Module`), its corresponding *domain* (`Landing Set`) and the *requirement* that led to this domain. `Computing Module` is the software machine that controls the Landing Set domain to ensure the requirement. Dividing a software purpose into a set of manageable and well-documented pieces makes is easier to comprehend a complex problem and to reuse pieces of the decomposition with the benefit of their context (domain and requirement).

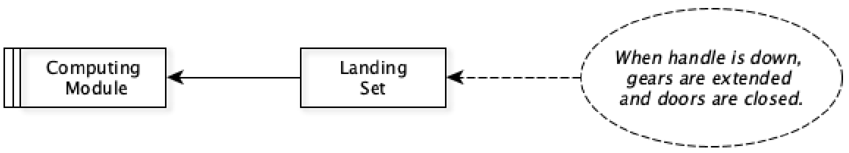

Computing Module
Landing Set
When handle is down, gears are extended and doors are closed.

Fig. 7. LGS Landing Set Problem Diagram

Assessing Problem Frames according to the criteria of section 3.3:

- *Scope*: Problem Frames focuses on the modeling of the system in its environment, treating both aspects as equally important.
- *Audience*: the notation, while graphical and intuitive, requires understanding the methodology.
- *Abstraction*: this approach is for requirements mostly, not influenced by implementation concerns, and is very abstract due to its specific notation.
- *Method*: the approach applies Jackson's methodology.
- *Traceability*: the approach focuses on the expression of dynamic behavior requirements and offers no specific support for traceability.
- *Coverage*: Problem Frames mainly address functional requirements.
- *Semantics*: While precise, the graphical notation has no formally defined semantics.
- *Tools*: no specific tools appear to be available.
- *Verifiability*: the approach is not designed for verification.

4.3.3 FSP/LTSA. "Process algebras", which provide a formal basis for describing interactions between processes, have influenced several methods. The original algebras were Hoare's CSP (Communicating Sequential Processes) [49] and Milner's CCS (Communicating Sequential Processes, later extended to cover mobile agents in the π-calculus [79]). FSP [69] proceeds from both.

The basic unit of all process algebras is a concurrent process which can communicate with others through input and output ports. For requirements, communication can model interaction both within the system and with the environment. Specifications in this style can formally express such essential temporal properties as liveness, safety, progress and fairness, and use supporting tools to verify them.

For FSP the supporting tool is LTSA [69]. From an FSP model, LTSA generates a Labeled Transition System (LTS), suitable for automated analysis and animation. LTSA is *compositional*, meaning that it is possible to model the components of a system separately then, with process calculus mechanisms, their composition. In addition to helping the modeling process, LTSA's compositionality benefits the verification process: one may verify safety and liveness properties by model-checking individual components then their composition.

FSP can take advantage of compositionality to model the LGS example as the parallel composition (expressed through the `||` operator) of two distinct processes:

```
||LGS = (LGS_BEHAVIOR || CONTROL_HANDLE)
```

CONTROL_HANDLE specifies how the state of the handle may change, and LGS_BEHAVIOR how the LGS reacts to these changes.

The history of a process's execution, called a *trace*, is defined by a sequence of transitions each executed in response to a certain event from the *alphabet* of the process. Processes such as the above two interact through events in the intersection of their alphabets, such as "up" and "down".

The concept of *fluent* serves to express that a process may be in a certain state which it can only leave through specific transitions. For example, specifying

```
fluent HANDLE_IS_DOWN = <{down}, {up}>
fluent HANDLE_IS_UP = <{up}, {down}>
```

specifies the notion of the handle staying up and down in "*the handle has been pushed up and stays up*", from requirements R12bis and R22, and the dual property from R11bis and R21. Only the specified transitions can, in each case, set and unset the fluent.

Here are further fluents for LGS, some with more than one setting or resetting event:

```
fluent DOOR_IS_CLOSING = <{start_closing}, {end_closing, open}>
fluent DOOR_IS_CLOSED = <{end_closing}, {open}>
fluent GEAR_IS_EXTENDING = <{start_extension}, {end_extension, start_retraction}>
fluent GEAR_IS_EXTENDED = <{end_extension}, {start_retraction}>
fluent GEAR_IS_RETRACTING = <{start_retraction}, {end_retraction, start_extension}>
fluent GEAR_IS_RETRACTED = <{end_retraction}, {start_extension}>
```

Some of the new events, such as start_closing, do not immediately reflect an event expressed in the informal LGS specification, but are artifacts for expressing timing properties in the FPS framework.

LTSA also supports assertions to express properties of the system. In the LGS example:

```
assert R21 = [] ([] HANDLE_IS_DOWN -> [] ! GEAR_IS_RETRACTING)
assert R22 = [] ([] HANDLE_IS_UP -> [] ! GEAR_IS_EXTENDING)
assert R11bis = [] ([] HANDLE_IS_DOWN -> <> [] (GEAR_IS_EXTENDED && DOOR_IS_CLOSED))
assert R12bis = [] ([] HANDLE_IS_UP -> <> [] (GEAR_IS_RETRACTED && DOOR_IS_CLOSED))
```

with the following syntax for operators of boolean and temporal logic: ! is negation, && is conjunction, -> is implication, [] is "always" and ⟨⟩ is "eventually".

The specification as given so far would not verify because of FPS's default assumption of equal priority of all applicable transitions, to ensure fairness. In R21, [] HANDLE_IS_DOWN cannot hold because an "up" event will eventually invalidate this fluent; similarly for R11bis. To resolve such situations, LTSA provides the *lower priority* operator, written ⟩⟩. We can use it to resolve the conflict in favor of "down" by lowering the priority of "up", rewriting the definition of the system as

```
        ||LGS = (LGS_BEHAVIOR || CONTROL_HANDLE) >> {up}.
```

and use the following auxiliary assertion to check the effect:

```
    assert EVENTUALLY_ALWAYS_DOWN = <> [] HANDLE_IS_DOWN
```

Then R21 and R11bis will verify but not R22 and R12bis, for which we would need instead to decrease the priority of "down" and assert EVENTUALLY_ALWAYS_UP. It is a characteristic of LTSA that in such cases one cannot verify both sets of assertions under the same conditions.

Appendix A (in the addendum [16]) contains the complete FSP model for the LGS example. The reader may input it "as is" into the LTSA analyzer.

Assessing LTSA according to the criteria of section 3.3:

- *Scope*: the approach can cover both system and environment aspects of requirements.
- *Audience*: the FSP notation is mathematically formal and requires the corresponding qualification both from the specifiers and the readers.
- *Abstraction*: the FSP abstraction is only suitable for specifications and requirements modeling.
- *Method*: the approach does not assume a particular requirements engineering method.
- *Traceability*: the approach focuses on model-checking FSP specifications, so it offers no specific support for traceability.
- *Coverage*: LTSA can specify and model-check liveness, safety, progress and fairness.
- *Semantics*: the FSP semantics is rigorously defined [69].
- *Tools*: the LTSA tool supports model-checking FSP specifications, execution of specifications, and graphical simulation.
- *Verifiability*: LTSA supports verification of liveness, safety, progress, and fairness properties.

4.4 Other mathematical frameworks

Requirements can use mathematical theories other than graphs and automata, for example set theory, the ultimate basis for approaches discussed below: Event-B (section 4.4.1), Alloy (4.4.2) and FORM-L (4.4.3). Other important mathematics-based approaches include VDM (Vienna Development Method), and Tabular Relations (4.4.A and 4.4.B in the addendum [16]) as well as TLA+ [61].

4.4.1 Event-B. Event-B [1] is a formal method for system-level modeling and analysis. Modeling proceeds by specifying the system's state in terms of sets and functions, and specifying state transformation in terms of events. The mathematical basis is set theory complemented by *refinement*, a mechanism for turning a description of a system at a certain level of detail into a new one that remains consistent with it but includes more details. To ensure this consistency, refinements must preserve the *invariants* from the original description, while possibly adding new ones. The preservation of invariants must be proved mathematically, with the help of the supporting tools.

Event-B has been used in industrial projects requiring proofs of correctness, particularly in the transportation and aerospace fields and in business management [107], and in combination with other approaches such as Problem frames (section 4.3.2), in the automotive industry [42].

Fig. 8 shows an Event-B machine modeling part of the LGS.

The machine "sees" the LGS context model, meaning that it inherits ground elements: constants, carrier sets and axioms governing them. Its three variables capture the state of the LGS. The first three invariant assertions define the variables' types in terms of the carrier sets. They are axioms; in contrast, the next two, $R21$ and $R22$, marked *theorem*, are proof obligations, capturing the eponymous LGS requirements. The *Initialisation* event assigns initial values to the variables. A complete machine model of the LGS would contain more events defining the behavior of the LGS controller.

Event-B does not natively support specification of temporal properties, which is why we could not use it to capture requirements R11bis and R12bis.

```
MACHINE LGS_Machine
SEES LGS_Context
VARIABLES
      gear_status
      door_position
      handle_position
INVARIANTS
      inv1: gear_status ∈ GearStatus
      inv2: door_position ∈ DoorPosition
      inv3: handle_position ∈ HandlePosition
      R21: ⟨theorem⟩ (handle_position = handle_down) ⇒ (gear_status ≠ gear_retracting)
      R22: ⟨theorem⟩ (handle_position = handle_up) ⇒ (gear_status ≠ gear_extending)
EVENTS
Initialisation
   begin
        act1: gear_status := gear_extended
        act2: door_position := door_closed
        act3: handle_position := handle_down
   end
END
```

Fig. 8. Fragment of the Event-B machine that models the LGS.

The Event-B refinement process incrementally add details or extensions to the model, requiring a proof of preservation of the invariants by the instructions in the events. The Rodin platform [31] partly automates the proof process, with the possibility of manually introducing custom proof tactics when the tool cannot discharge a proof completely automatically.

Assessing Event-B according to the criteria of section 3.3:

- *Scope*: the approach makes it possible to model both a system and its environment.
- *Audience*: the notation requires familiarity with classical set theory; while the concepts are elementary, they exclude stakeholders who feel uncomfortable with mathematics.
- *Abstraction*: through successive refinement, the approach covers the full spectrum from very abstract and partial models to detailed final models ready for translation into an implementation.
- *Method*: Event-B is a method based on successive refinements proved invariant-preserving. It does not cover the entire process of requirements engineering (e.g. how to obtain requirements from stakeholders), only refinement and proof.
- *Traceability* is not a particular focus, although it is possible to trace the identifiers used throughout the refinement process.
- *Coverage*: Event-B has no particular mechanism for modeling non-functional requirements.
- *Semantics*: the behavioral semantics of Event-B refinement has been described in [110].
- *Tools*: tools, particularly in the Rodin environment [31] support refinement and proof.
- *Verifiability*: requirements expressed in Event-B are verifiable through the proof process which accompanies refinement: the description at every step of the refinement must be proved consistent with the description at the preceding, immediately higher level. The industrial projects involving verification, mentioned above, applied this process.

4.4.2 Alloy. Alloy [57] is a declarative modeling language based on first-order logic for expressing the behavior of software systems. Alloy is a subset of the Z set-theory-based specification language [2], also the starting point for Event-B.

Alloy has spurred a significant research community and a number of applications. In [136], Zave used Alloy to provide the first specification of correct initialization and operations of the Chord ring-maintenance protocol [116], through a formal model which she proved to be correct.

In applying Alloy to the LGS example, we note that Alloy has no native mechanism for expressing properties such as "*handle remains in the DOWN position*" in R21. Alloy, however, can specify state transitions. Consider this rewrite of R21:

R21 If the landing gear command handle *is* down, the gear is not retracting.

with "remains" changed to "is". As an implication with a weaker antecedent, this new R21 is stronger than the original. It can be expressed in Alloy:

```
R21: check {
    all lgs, lgs': LGS | ((lgs.handle in Down) and (lgs'.handle in Down) and Main [lgs, lgs'])
        implies (lgs'.gear not in Retracting)
} for 5
```

where *Main* is a predicate that yields "true" if and only if *lgs* and *lgs′* represent two consequent states of the LGS. The number 5 is the size of the search space, serving as bound for Alloy's use of bounded model checking for verification.

The absence of temporal operators in Alloy similarly suggests rewriting R11bis as:

R11bis If the handle remains down, *three transitions of the LGS will suffice to ensure that* the gear is extended and the door closed.

This form (where the italicized expression replaces "*eventually*") can be expressed in Alloy:

```
R11bis: check {
    all lgs1, lgs2, lgs3, lgs4 : LGS |
        (((lgs1.handle in Down and lgs2.handle in Down and lgs3.handle in Down and lgs4.handle in Down)
              and
        (Main [lgs1, lgs2] and Main [lgs2, lgs3] and Main [lgs3, lgs4]))
            implies (lgs4.gear in Extended and lgs4.door in Closed))
} for 5
```

Three transitions involve four different states, which is why the assertion declares four variables of type "LGS" under the universal quantifier.

Assessing Alloy according to the criteria of section 3.3:

- *Scope*: Alloy can only specify the target system.
- *Audience*: the Alloy notation is formal and requires the corresponding mathematical qualification from both specifiers and readers.
- *Abstraction*: Alloy can specify both requirements and implementations.
- *Method*: Alloy does not assume or promote a particular requirements engineering method.
- *Traceability*: focused on specifications and their verification, Alloy does not cover traceability.
- *Coverage*: Alloy supports specifying and model-checking behavioral specifications, without consideration of non-functional properties.
- *Semantics*: the Alloy semantics is rigorously defined in [57].
- *Tools*: the Alloy analyzer supports model-checking of Alloy specifications, execution of specifications, and graphical simulation.

- *Verifiability*: Alloy models can be analyzed for consistency with counterexample-guided model-checking, and for correctness of predicate-logic assertions with bounded model-checking.

We have specified and verified a complete Alloy LGS example: Appendix B in the addendum [16].

4.4.3 FORM-L. FORM-L [91] extends MODELICA [81], an object-oriented notation for modeling the behavior of physical *systems.* FORM-L results from the MODRIO project (MOdel DRIven physical systems Operation), which improved MODELICA by adding the modeling of assumptions on the *environment.* Fig. 9 shows an example FORM-L specification.

```
requirement r11 is
    after (lgsHandle becomes down)
        and (not failure)
        and not (allGearsDown and allDoorsClosed)
    within 15*s
    check (allGearsDown and allDoorsClosed) becomes true
         or lgsHandle becomes up
         or failure becomes true;
```

Fig. 9. Example of FORM-L requirement

FORM-L addresses the early stage of system development, with a level of detail sufficient to support some early validation through model-checking using the Stimulus tool [115]. Thanks to its formal semantics the FORM-L requirements can be verified by model-checking.

Assessing FORM-L according to the criteria of section 3.3:

- *Scope*: the approach can cover both system and environment aspects.
- *Audience*: the FORM-L notation is formal and requires the corresponding mathematical qualification from both specifiers and readers.
- *Abstraction*: FORM-L can specify detailed design as well as goals and requirements.
- *Method*: the approach does not assume any particular requirements engineering method but the main definition steps focus on: Goals, Requirements, Specification, Design.
- *Traceability* is among the goals but with no supporting mechanisms so far.
- *Coverage*: FORM-L covers both functional and non-functional properties.
- *Semantics*: the FORM-L notation is in the process of being formalized by providing a formal semantics to a kernel set of FORM-L concepts.
- *Tools*: only tools internal to EDF, the organization that developed FORM-L, support code generation and simulation.
- *Verifiability*: the approach supports simulation.

4.5 Seamless, programming language based approaches

The approaches reviewed so far focus on the requirements task, separately from others such as design and implementation. *Seamless* approaches emphasize instead its commonality with the rest of the software process and go so far as to use a programming language as a notation for requirements.

Two such approaches (each involving some of the authors of this survey) are the multirequirements method (section 4.5.1) and a refinement of it, Seamless Object-Oriented Requirements (4.5.2). They are assessed jointly in section 4.5.3.

Their common underlying idea is the "Single Model Principle" described by Paige and Ostroff [97] and going back to [73]. It postulates a fundamental unity in the software process, from requirements to design, implementation, validation and maintenance, and sees gaps between these tasks ("impedance mismatches") as threats to quality; to avoid them, it advocates a seamless process with a consistent set of notations and tools throughout. Seamlessness is, in particular, a way to address change, making it possible, instead of having to modify several artifacts, to modify a single one expressed in a language that captures the common semantics [103].

4.5.1 Multirequirements. The multirequirements [75] method promotes the development of a requirements document as an interweaving of several descriptions, such as the following three:

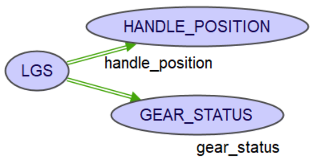
- Natural language text.
- A graphical notation such as UML or similar.
- A formal version, expressed not in a special formal-specification notation but in a programming language with enough support for abstraction and semantics, such as Eiffel.

The three interwoven descriptions complement each other, each contributing its best traits: natural language for context and explanations; graphics for high-level views; and the formal variant for precision. The argument for using a programming language as formal notation is that programming languages are defined with a precise semantics; for requirements purposes, the approach ignores imperative constructs (assignment, control structures, variables, the notion of state) but retains the structuring constructs such as classes, genericity, information hiding and inheritance. The multirequirements method as developed so far uses Eiffel as the programming language, relying on its abstraction and modularization mechanisms and its facilities for expressing semantic properties through "Design by Contract" (DbC [73]) constructs: preconditions, postconditions, class invariants.

The following description of part of the landing gear system illustrates the approach.

(1) The state of the /LGS/ system is defined by the position of the handle in the cockpit and the current status of the gear itself:

```
class LGS
feature
    handle_position: HANDLE_POSITION
    gear_status: GEAR_STATUS
end
```



(2) Whenever the handle is in the down position, the gear should not be in the retraction state. We label this requirements as R21:

```
class+ LGS
invariant
  r21: (handle_position = down) implies (gear_status ≠ retracting)
end
```

The extract includes elements of all three kinds: English text, class text in Eiffel, graphical diagrams (here automatically generated by EiffelStudio from the class text). Conventions support traceability between these complementary forms of description: an occurrence of /C/ in the English text (here /LGS/ in paragraph (1)) is a reference to a class of name C in the formal and graphical elements. The other way around, a tag in the formal text, such as `r21:` in the invariant, is a reference to a paragraph (not shown) of the text. To facilitate incremental construction of requirements, the notation "class+", as in paragraph (2), indicates an addition to a class partially specified before, here class LGS, with the expectation that tools can provide the full description combining all these increments.

[90] presents a detailed application of multirequirements to a cyber-physical system example.

4.5.2 Seamless object-oriented requirements. The basic Design by Contract mechanisms (preconditions, postconditions, class invariants) cannot handle temporal logic properties such as R11bis and R12bis. The SOOR method [85] adds expressive power for capturing and verifying such properties, as well as support for environment properties and requirements reuse. It takes advantage of imperative features of the programming language to write *specification drivers* [88] which describe properties of *several* operations of a class and also serve as "parameterized unit tests" [124]. The corresponding auxiliary routines rely on pre- and postconditions to specify and verify behavior under different environment assumptions [88]; their loops rely on loop invariants and variants to specify and verify temporal and timing properties [89]; and the classes rely on object-oriented techniques of genericity and inheritance to support requirements reuse [86]. DbC elements make it possible to prove properties of requirements with the AutoProof verifier for Eiffel [125].

The SOOR representation of the R12bis requirement takes the following form:

```
class
    R12_BIS
inherit
    RESPONSE_GLOBAL [LGS, GEAR_EXTENDED_DOOR_CLOSED_OR_ELSE_HANDLE_DOWN, HANDLE_UP]
    LGS_REQUIREMENT
end
```

The `RESPONSE_GLOBAL` class captures the following LTL pattern of the same name [30]:

$$\Box(P \Rightarrow \Diamond S) \tag{1}$$

`RESPONSE_GLOBAL` is a general reusable pattern, taking advantage of inheritance and the generic parameters to express that *when the HANDLE is pulled UP, the GEAR will eventually be seen EXTENDED and the DOOR will be seen CLOSED; OR ELSE, we conclude that the HANDLE is pushed back DOWN.* Reusing the "specification driver" of the general class avoids writing an LTL formula manually. The specification can be used for either testing or proofs, in the latter case through AutoProof, which will only accept it if classes (here `LGS`) have correct and strong enough contracts.

Approaches that ensure conformance of natural language requirements to predefined templates [9, 34] would facilitate application of the SOOR approach in practice.

4.5.3 Assessment. Assessing both the multirequirements and SOOR approaches according to the criteria of section 3.3:

- *Scope*: the approaches focus on system aspects. Environment properties could in principle be modeled in a similar way, but that aspect remains to be developed.
- *Audience*: the three-layer representation yields complementary specifications, readable by different stakeholders such as requirements analysts, software developers, testing engineers.
- *Abstraction*: the use of programming-language notation makes it possible to cover the full spectrum from the most abstract requirements to the most concrete aspects of implementation.
- *Method*: the approaches rest on a strong methodological basis, integrating the principles of object-oriented analysis [73].

- *Traceability*: the Eiffel Information System (EIS) tool supports traceability between multirequirements and implementations. The EiffelStudio IDE (development environment) supports traceability of all texts in the language, whether for requirements, design or implementation.
- *Coverage*: the approaches focus on functional requirements.
- *Semantics*: the semantics of multirequirements comes from the semantics of contracts in [74].
- *Tools*: EIS partially supports the multirequirements process; AutoProof supports SOOR.
- *Verifiability*: in the formal layer, requirements can be verified through tools such as AutoProof and AutoTest.

5 SUMMARY OF RESULTS AND DISCUSSION

To draw conclusions from the present study, we first list its limitations (section 5.1), then present a summarized table of results (5.2) and explore conceptual questions raised by the analysis:

- Should the elicitation process start with an informal or semi-formal notation (section 5.3)?
- Is a seamless approach better or worse than mixing formal and semi-formal notations (5.4)?
- Are there other ways to combine formal and informal approaches (5.5)?
- What are the merits of natural language and graphical notations for requirements (5.6)?
- What is the current state of tool support for requirements engineering (5.7)?
- What is the current state of education in formal approaches to requirements (5.8)?

5.1 Limitations

While we have striven to make this review comprehensive, the following decisions may affect the generality of its results:

- The choice of a running example, the Landing Gear System. An alternative would have been to include a multitude of small examples, each chosen to make the corresponding approach shine, whereas the LGS may be more suitable to some than to others. The case for a single example is clear: to permit a significant comparison of the approaches.
- The nature of that example, a reactive system. An alternative would have been an enterprise-style system (such as accounting and Web content management). The case for a reactive system is that such applications are among the hardest to build, so they are likely to test to their limits the advantages and deficiencies of the methods surveyed.

5.2 A summary of the results

Table 2 presents the key conclusions in tabular form, ordered by category (from section 4), then alphabetically within each category. The caption explains the conventions.

Table 2 may serve as a "Swiss Army knife" for choosing the right method(s) for a given problem. No method has all the desired characteristics. The table reveals gaps in the different methods and may help researchers identify fruitful research projects to fill these gaps.

5.3 Formal vs. informal notations for the elicitation process

A requirements document should be both precise and understandable. These objectives can conflict with each other. Among the approaches surveyed, formal methods favor precision at the possible risk of obscurity for non-experts; others, particularly natural-language-based and graphical, favor understandability, at the possible risk of renouncing precise semantics.

The issue of informality versus formality in the process of requirements engineering is not new. The author of [55] concludes that techniques providing a high degree of guidance and process description are critical to achieve successful results. Van Lamsweerde, in [126], concludes that higher-level abstractions for requirements specification and analysis are critical success factors.

	Scope	*Audience prerequisites*	*Level of abstraction*	*Associated method*	*Traceability support*	*Non-functional requirements support*	*Semantic definition*	*Tool support*	*Verifiability*
Requirements Grammar	B	N	H	✘	✘	✓	✘	✓	✘
Relax	B	N	H	✘	✓	✓	✓	✓	✓
Stimulus	B	N	H	✘	✘	✘	✓	✓	✓
NL to OWL	B	S	H	✓	✓	✓	✓	✘	✘
NL to OCL	S	S	B	✓	✘	✘	✓	✓	✓
NL to STD	B	S	H	✓	✘	✓	✓	✘	✓
Reqtify	B	N	L	✘	✓	✓	✘	✓	✘
KAOS	B	S	B	✓	✓	✓	✓	✓	✘
SysML	S	S	H	✘	✓	✓	✘	✓	✘
URN	S	S	H	✓	✓	✓	✘	✓	✘
URML	B	S	H	✘	✓	✓	✓	✓	✘
Statecharts	S	S	H	✓	✘	✘	✓	✓	✓
Problem Frames	B	S	H	✓	✘	✘	✘	✓	✓
FSP/LTSA	B	S	H	✘	✘	✓	✓	✓	✓
Petri Nets	S	S	H	✘	✘	✘	✓	✓	✓
Event-B	B	F	B	✓	✘	✘	✓	✓	✓
Alloy	S	F	H	✘	✘	✘	✓	✓	✓
FORM-L	B	S	H	✘	✘	✘	✓	✓	✓
VDM	B	F	H	✓	✘	✘	✓	✓	✓
Tabular Relations	B	M	L	✘	✓	✘	✓	✘	✓
Multirequirements	B	S	B	✓	✓	✘	✓	✓	✓
SOOR	B	S	B	✓	✓	✘	✓	✓	✓

Table 2. Assessment summary. *Scope* (system versus environment): either S (the approach can be only used to model the system) or B (the approach can be used to model both system and environment). *Audience prerequisites*: one of F (formal methods background), M (general mathematical knowledge), S (specific training required other than F and M), N (no particular background expected). *Level of abstraction*: one of L (low), H (high), B (both).

Formal methods do have uses in industry, as attested by a number of significant success stories, but such cases remain a minority; most projects still rely on natural language. This survey may provide some insights on how they might benefit from formal ideas.

The usual argument against formal methods is that they are hard to understand. It has limits, however. Stakeholder comfort is a concern, but has to be matched against considerations of quality of the final system ("*will the plane crash?*"). Only a formal version can serve as a basis for a cohesive and unambiguous statement of client's needs, solid enough in principle to serve as a legal contract.

This counter-argument (in favor of formal methods) has its own limits. The rigor and precision of formal methods is not an excuse for ignoring the need to understand what stakeholders want. Even a system that has been formally "proved correct" has only been proved to *satisfy a given specification.* However sophisticated the proof, if the specification does not reflect the stakeholders' desires, the system is in fact incorrect for all practical purposes. This observation is not just theoretical: numerous studies, most spectacularly by Lutz about NASA software [68], point to system failures resulting not from a technical error but from a bad understanding of user needs.

Any successful requirements method, formal or not, must provide ways to understand and record stakeholders' intent. Unlike what a simplistic view might suggest, this process is not just a one-shot "requirements elicitation" phase but often, in practice, an iterative negotiation. Informal and graphical approaches have an advantage here since they are easy to explain to a broad range of stakeholders. To succeed on a large scale, formal methods and tools must provide similar mechanisms to interact with experts in the problem domain who are not experts in requirements. The discipline of requirements engineering traditionally recognizes (see textbooks such as [131] and [62]) the need for "requirements engineers", also called "business analysts", who help translate needs as expressed by stakeholders, particularly "domain experts", into bona fide requirements. To be successful for requirements elicitation, any formal method must develop its own cadre of such mediators, who possess both expertise in the method and an ability to relate to ordinary project stakeholders. Proponents of formal methods often complain about the reluctance of stakeholders to use mathematical reasoning. Complaining does not lead anywhere and deflects from the formalists' own responsibility: never to start a formal-method-based requirements process without the right investment in requirements engineers who will translate back and forth between formal and informal views.

The experience of the database community may provide guidance. In database design, the initial phase (before the switch to an implementation that often uses the relational model, another notation with a solid mathematical basis [22]), typically relies on a graphical semi-formal notation such as entity-relationship diagrams [21], possessing a precise semantics but intuitive enough for initial design and explainable to non-expert stakeholders. This experience shows that, with a proper process in place, there is no reason to fear systematic rejection of formal or semi-formal approaches.

5.4 Role of seamlessness in bridging the formal-informal gap

The dominant view in software engineering is that requirements and code are two fundamentally different products, to be handled through different methods, tools and languages. The drawback is the risk of divergence: software evolves, both on the requirements side and on the code side, and it is difficult to maintain consistency, as expressed by the concern for traceability.

An alternative approach, discussed in section 4.5, uses seamless development, relying on a single set of concepts and a single notation — a programming language — throughout.

The idea of using a programming language for requirements often triggers the reaction that programming languages are implementation-oriented and usually imperative, jeopardizing the necessary focus of requirements on "what" rather than "how" — the Abstraction criterion of section 3.3. Modern programming languages, however, are not just about implementation; they provide powerful structuring mechanisms such as the notions of module/package, class, inheritance, information hiding, interface and genericity, applicable to the modularization of requirements. Ordinary mathematical notation is not designed for the description of large systems. Seamless approaches tap into the structuring mechanisms of programming languages for these large-scale structuring needs.

Another goal of using a programming language is to narrow gaps ("*impedance mismatches*") between requirements and other steps. If everything is in a single notation it is easier, according to advocates of seamlessness, to keep the various products (requirements, design, implementation) in sync, with benefits for traceability, debugging and maintenance. ("Single Model Principle", 4.5.)

More work, in particular empirical, remains necessary to assess this thesis:

- Does seamlessness clarify requirements for stakeholders with widely different backgrounds?
- What are the concrete traceability benefits?
- How much does seamless development reduce documentation overhead?

- How much does seamlessness support requirements maintenance and reuse?
- In a seamless approach, do IDEs provide enough tool support, or do requirements still call for specific tools?
- How can the processes of requirements and implementation reinforce each other?

5.5 Other ways of combining formal and informal elements

In addition to the work on seamlessness and multirequirements, a number of authors have explored, in some cases through empirical studies, how to combine formal and informal elements. Golra et al. [43] — also using LGS as an example — present Model Federation, an approach for incremental co-development of informal requirements and formal specifications, with tool support for fine-grained traceability between the two sides. The results of their experiment suggest that the approach facilitates early validation and verification of requirements, and yields formal specifications that are consistent with these requirements and usable as the starting point for implementation and verification.

Rodrigues et al. [25] provide empirical evidence from industry that agile teams can have a positive perception of formal approaches to requirements. The participating teams found that the use of a formal approach — specifically, Z — yielded positive effects: streamlining the requirements specification process, and faster understanding of the requirements. These results belie the common perception that using formal methods in requirements involves a long learning phase: the team was able to apply the offered formal method after a one-hour training session.

Fraser et al. [37] emphasize the importance of round-trip requirements engineering between natural language based and formal notations, illustrating the argument through the combined use of the Structured Analysis natural-language-based approach and the formal VDM approach (section 4.4.A in the online addendum [16]). The article presents two approaches (manual and automated) to producing and subsequently refining VDM models from the Structured Analysis descriptions.

5.6 Respective merits of natural language and graphical notation

In the discussion of how to make requirements understandable and expressible by various kinds of stakeholders, formal-versus-informal is not the only relevant criterion. Another opposition is textual versus graphical notations. The two distinctions are in fact orthogonal, as all four combinations exist:

- Informal specifications can be expressed in (textual) natural language, but they can also be graphical, in part or in full.
- Formal specifications, often based on a textual mathematical notation, can also be expressed graphically. For example, the presentations of Petri nets typically use a graphical form.

While critics of formal requirements emphasize that stakeholders without a strong software or mathematical background can react negatively to formal texts, informal requirements are not necessarily the solution either: a long and verbose informal text can be just as off-putting.

In contrast, graphical notations can make complex structures readily understandable. They can express spatial relations in an intuitive manner, which text cannot. Among approaches covered in this survey, SysML, KAOS and i* are examples (of various degrees of formality) that strongly and effectively rely on graphical notations. Another example, from the database community, was cited above: entity-relationship diagrams.

Diagrams have clear advantages and limitations. A picture, it is said, is worth a thousand words. But it cannot carry the details of all these words. Graphical presentations are good at describing the overall scheme of a system — what, with a revealing choice of words, is called “the big picture”. For example, we can express graphically that an airplane guidance system has a component to control the trajectory and another to monitor it and raise alarms. Graphics is not, on the other hand, the best way to state the exact conditions (aircraft's altitude and angle) that will trigger an alarm. It should also be noted that the graphical nature of a notation does not guarantee its wide accessibility: some graphical notations, such as wiring diagrams (in electronics) and rail plans, are intelligible to specialists only.

As with formal versus informal approaches, textual and graphical notations are best viewed as complementary techniques, outside of any dogma, each to be used when and where it is the best way of specifying a given requirement element. "Multirequirements" (4.5.1) combine all three kinds.

5.7 Current state of tool support for requirements engineering

Engineering the requirements of today's complex and ambitious systems cannot be a purely manual process. Any realistic solution requires tool support. Modern requirements methods indeed come with tools, as cited in the previous sections. Graphical user interfaces, for example, are available in tools associated with methods ranging from the most informal to the fully formal, such as Reqtify [118] for DOORS (section 4.2.1), Objectiver [106] for KAOS (4.2.2), Enterprise Architect [119] for SysML (4.2.3), Overture [101] for VDM (4.4), AutoProof [125]) for multirequirements in Eiffel (4.5.1). Beyond user interfaces, however, what matters is how these tools help the requirements process. In particular:

- Which parts of requirements engineering do they facilitate?
- How do they help achieve the core goal, requirements quality?

A 2011 survey of 94 requirements tools [18] found that the emphasis was on modeling (42% of the tools) and requirements management (39%). This trend has continued. A commercial site listing the most widely used requirements tools [17] suggests that the principal functions of today's tools are requirements elicitation, change tracking and traceability. Reqtify, for example, provides functionalities to trace requirements and link requirements to artifacts of various kinds. The tools for KAOS [106] and i* focus on requirements elicitation. In addition to elicitation, tools for SysML [119] and URML help organize requirements into models and hierarchies, and the refinement process. All such tools support the requirements process and its integration with the rest or the development cycle but do not support the processes of formalization and deductive reasoning.

Approaches based on seamlessness and the Single Model Principle (section 4.5) make it possible to rely on program-proving tools such as AutoProof to prove not only correctness properties of the future program (meaning its conformance to requirements) but also, at the requirements stage, consistency properties of the requirements themselves, independently of any future implementation. In addition to enabling proofs, such an approach may benefit from modern tools for automated *testing*, such as AutoTest [76], Pex [123] or AxiomMeister [122]. Whether proof- or test-oriented, these tools need contracts as a basis for formal verification.

We note here a contribution of seamless development to the general discussion of tools for requirements, including those following other approaches: for maximum effectiveness *tools supporting requirements should support more than requirements*. Consider the example of traceability. Some requirements tools have very good support for traceability between requirement elements. But traceability is also about tracing relations between requirements and other artifacts, particularly design, code and tests [15]. (Traceability here involves detecting the consequences of a requirements change on all such possibly affected artifacts, and the other way around.) A tool that focuses just on requirements will not address this critical need. The future, we believe, lies in integrated tools that capture, along with requirements, all other products of software engineering.

Open questions in the area of requirements tools include:

- Is there a general pattern for textual requirements, or does every domain area requires its own?
- Can tools help measure the quality of requirements?
- Should tools provide (in “multirequirements” style) different viewpoints tuned to each category of stakeholders?
- Can functioning code be generated automatically from formal requirements? Should it?

5.8 Current state of education in formal approaches to requirements

Interest in formal approaches to express requirements and, more generally, to design software has been progressively growing for the last few decades. Consequently, many educational institutions complemented their research effort with a pedagogical effort in software engineering curricula [66]. This synergy, combined with the increasing interest of industry in formal approaches, trained a generation of students capable of developing formal thinking since the early stages of their professional career, and bringing this attitude to their job environments. The literature on pedagogical aspects of formal methods is vast [28, 38], in particular on approaches to software engineering courses built on a strong mathematical basis [39]. Some articles document resistance to the deployment of such approaches and investigate the issue of motivation [96, 105].

The general-purpose requirements formalisms are not often used to teach requirements engineering. If SysML is used, it is mainly as part of the broader scope of education in model-based software engineering (MBSE) [93]. Other approaches such as KAOS and i* are essentially used at the requirements elicitation step [84], notably by students having no knowledge in formal methods [26], or to complement formal methods by domain-specific ones [56]. For example, a workshop series dedicated to i* teaching has been running since 2015 (iStar@CAISE2015, iStar@ER2017).

Natural-language-based approaches avoid the difficulty of teaching a new formalism. Most requirements engineering courses likely start with natural-language requirements. Publications such as [46, 139] emphasize the difficulty of teaching requirements elicitation and propose solutions. The difficulties, however, are not really due to natural language per se, but to the challenge of achieving requirements qualities such as completeness and consistency; and the solutions — such as as the practices recommended in [131] — are correspondingly techniques for writing good requirements, largely applicable regardless of the requirements approach and notation.

Graphs and automata are popular with students [8, 44]. While mathematical in nature, they are graphical and easy to understand and produce. This user-friendliness can lead to misunderstanding and errors because of the lacks of clear execution semantics, as reported by well-known articles [24, 36, 129]. In this respect, recent advances in executable semantics and tool animation (as discussed in section 5.7 above) may hold promises for education.

[19] describes experiences in teaching formal methods, in particular JML [64] and Event-B (section 4.4.1). [63] presents the design and delivery of courses aiming at developing skills in model construction and analysis by use of notations such as VDM-SL and VDM++. They address the motivation problem by using examples from industrial projects and an industrial-strength tool set.

[5] reports on the teaching of concurrency theory and FSP/LTSA. Among all formalisms for concurrency, CSP has enjoyed widespread applications, both industrial and educational [108].

[120] presents a more interactive way of introducing formal specifications, relying on the design of an online tutorial to help students transition from Z to Alloy, the latter considered more practical thanks to the Alloy Analyzer.

Seamless, programming-language-based approaches to requirements are not yet widespread, so there is little empirical data available on their suitability for teaching. But a few observations are possible. Since Design by Contract (DbC) is the basis, the transition to a similar approach for requirements will be easy to explain to an audience which has been introduced to programming using DbC; this is the case with both approaches mentioned in section 4.5: with the multirequirements method, requirements take the form of contracted excerpts from the final program; with SOOR, they take the form of contracted routines expressed in terms of the final program.

On education, a number of questions remain open:

- Are extensions to existing courses relying on DbC (Design by Contract) a suitable approach to teach both programming and formal methods with an emphasis on quality?
- Should software engineering courses emphasize seamlessness?
- Why is there so little emphasis on requirements in regular Software Engineering curricula?
- How much should formal notations appear in introductory courses?

6 CONCLUSION: THE ROLE OF FORMAL METHODS IN REQUIREMENTS

What degree of formality is appropriate in stating requirements for software systems? To shed light on that question, this survey has analyzed a wide range of techniques for expressing software requirements, with a degree of formalism ranging over a broad scale: from completely informal (natural language), through partially formal (semi-formal, programming-language-based), to completely mathematical (automata theory, other mathematical bases).

The question of formality has caused and continues to cause heated debates, almost as old as the very recognition of requirements engineering as a significant component of software engineering. In those discussions, the basic arguments for and against have not changed much over decades: inevitably, proponents of formal methods will point to the imprecision of natural language; just as inevitably, opponents will argue that formal texts are incomprehensible to many stakeholders. There is truth in such statements on both sides, but they cannot end the discussion. The detailed analysis and examples of this article should help reach better informed decisions.

As an example of the limits of classic but simplistic views, consider the "*many stakeholders will never be able to understand formal notations*" argument. In reality, no one can require that all stakeholders understand all details of requirements. There is no such rule in other engineering endeavors; the marketing manager for a car company, perhaps the primary stakeholder since what counts is how cars will sell, cannot understand all the engineering diagrams and technical decisions. Even if we limit our focus to software, many aspects of any sophisticated software system will remain impenetrable to *some* stakeholders: if the system's scope extends across many technical areas, as in the case of a banking system that touches on accounting, investment management, currency handling, fraud detection, data security and international transfers, no single stakeholder is an expert in all these disciplines.

One suspects that often the formal-is-hard argument is really formal-is-hard-for-*my-developers*. People who, for example, promote semi-formal Design by Contract techniques regularly hear such comments: this is too hard for our people, they would need retraining, or maybe they just do not have the right mathematical education. Such objections are worth considering, but raise questions: why do these concerns matter more than others such as verifiability of the requirements? Comparing again with other areas of engineering, a building contractor is unlikely to use as an excuse, if the circuits short, that he could not require his electricians to learn Watt's law.

Beyond simplistic arguments, we need a balanced view assessing formality against relevant *criteria* of quality. This is the focus of the present article: each of the reviewed approaches has been evaluated according to a set of criteria introduced in section 3.3. These criteria, while not the only possible ones, are intended to cover what is most important to the stakeholders of a system.

In light of that review, several observations serve as a counterweight to "formalism-is-hard":

- Stakeholders who do not understand a formal description of a system still need to understand many of its aspects (as the car marketer must understand what is new in the latest model).

The notion of *view* is useful here. Requirements for a system may have to rely on several views adapted to the needs of different stakeholders; examples include natural language, graphical, tabular views and of course a formal view, the appropriate one for stakeholders who need precision and must consequently be ready to deal with mathematical concepts. (Mathematics is not a torture imposed on innocent stakeholders; it is the only language with full precision and, as a consequence, the language of science.) From this perspective, a formal expression of the requirements does not compete with other variants, but complements and supports them.

- If requirements use multiple views, the question arises of how to guarantee that they are consistent; only a formally defined view has the rigor and precision needed to be usable as the basis to derive others. The multirequirements method (section 4.5.1) develops this idea further, proposing to write requirements in a combination of natural-language, graphical and formal notations, the formal one serving as the reference in case of ambiguity.
- For most practical uses, the level of mathematics actually required to understand formal descriptions, and even in many cases to write them is not particularly high. Many software engineers and other professionals have gone through science curricula in which they had to master challenging mathematical techniques, such as control theory and statistics. For most formal methods the underlying mathematics consists of basic set theory and basic logic in the form of propositional and predicate calculus. (Specifications of real-time systems may also use temporal logic, but it is a simple extension to logic and not hard to learn.) The difficulty is often apparent rather than real; a matter of attitude.
- Anyone working in software is used to highly formalized (although usually not mathematical) notations: programming languages, which leave no room for imprecision.
- Executable semantics for general techniques such as UML or SysML have enjoyed widespread use, showing that users *will* learn highly technical approaches when benefits are clear.

The last comment suggests a way to progress in the formal-versus-informal debate. Ideological discussions should yield to pragmatic considerations. Techniques will gain acceptance if they produce tangible benefits commensurate with the effort they require. Two crucial conditions are:

- Tools: even the most impressive method and elegant notation will not catch on without automated support. Good tools free programmers from mundane tasks, flag errors and inconsistencies, and scale up to large systems.
- Education: software engineering education often causes disconnects where it should emphasize synergy. Disconnect between requirements and subsequent tasks, particularly implementation. Disconnect between formal methods, often taught as a special advanced topic for theory-inclined students, and the practice of software engineering (section 5.4 discussed the arguments for a more *seamless* approach, which combines these tasks together).

Even these regrets about disconnects between courses rely on an optimistic assumption: that students take courses on requirements and courses in formal methods. It is in fact possible today to complete a computer science/informatics/software engineering curriculum without having had courses on both of these topics — or, in some cases, on either of them. Such curricula should be corrected: every software engineer needs to know about requirements engineering, the discipline of making sure that the implementation of systems meets the needs of their stakeholders and the constraints of the environment; every software engineer should know how to apply formal techniques when precision and guaranteed correctness are required; and every software engineer should know when and how requirements can benefit from formal methods.

Beyond their application to education, these observations describe the relationship between formal methods and requirements in software engineering. Formal methods are sometimes considered theoretical while requirements engineering is essential to the practice of software construction. For that practice, formal methods complement other requirements techniques rather than attempting to replace them. They can and should be a powerful help available to every requirements engineer or business analyst.

We hope that the present survey has demonstrated this potential contribution of formal methods to requirements. We also hope that it will contribute to expanding their role for the greater benefit of future software systems and the people who depend on them.

Acknowledgments

We are grateful to the authors of the surveyed approaches who took time to check our rendering of their approaches and the treatment of the LGS examples. (All responsibility for these descriptions remains ours.) They include in particular Jeff Kramer, who went so far as to develop his own LTS specification of the LGS example from which our specification benefited considerably, in addition to providing insightful comments, and Thuy Nguyen for his formalization of the LGS in Form-L. We are also grateful to the reviewers of the first version for detailed and important comments.

This research has been financially supported by the Ministry of Digital Development, Communications and Mass Media of the Russian Federation and Russian Venture Company (Agreement No. 004/20 dd. 20.03.2020, IGK 0000000007119P190002).

REFERENCES

[1] J.R. Abrial. 2010. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press.

[2] J.R. Abrial, S. A. Schuman, and B. Meyer. 1980. Specification Language. In *On the Construction of Programs.* Cambridge University Press, 343–410.

[3] S. Abualhaija, C. Arora, M. Sabetzadeh, L. C. Briand, and E. Vaz. 2019. A Machine Learning-Based Approach for Demarcating Requirements in Textual Specifications. In *RE.* IEEE, 51–62.

[4] D. Aceituna, H. Do, G. Singh Walia, and S.W. Lee. 2011. Evaluating the use of model-based requirements verification method: A feasibility study. In *EmpiRE.* IEEE Computer Society, 13–20.

[5] L. Aceto, A. Ingólfsdóttir, K. Guldstrand Larsen, and J.í Srba. 2009. Teaching Concurrency: Theory in Practice. In *TFM (Lecture Notes in Computer Science)*, Vol. 5846. Springer, 158–175.

[6] D. Amyot. 2003. Introduction to the User Requirements Notation: learning by example. *Comput. Networks* 42, 3 (2003), 285–301.

[7] P. Arcaini, A. Gargantini, and E. Riccobene. 2017. Rigorous development process of a safety-critical system: from ASM models to Java code. *Int. J. Softw. Tools Technol. Transf.* 19, 2 (2017), 247–269.

[8] P. Arnoux and A. Finkel. 2010. Using mental imagery processes for teaching and research in mathematics and computer science. *Int. J. of Mathematical Education in Science and Technology* 41, 2 (2010), 229–242.

[9] C. Arora, M. Sabetzadeh, L. C. Briand, and F. Zimmer. 2015. Automated Checking of Conformance to Requirements Templates Using Natural Language Processing. *IEEE Trans. Software Eng.* 41, 10 (2015), 944–968.

[10] S. Bechhofer. 2018. OWL: Web Ontology Language. In *Encyclopedia of Database Systems (2nd ed.).* Springer.

[11] B. Berenbach, F. Schneider, and H. Naughton. 2012. The use of a requirements modeling language for industrial applications. In *RE.* IEEE Computer Society, 285–290.

[12] D. Bjørner and C. B. Jones (Eds.). 1978. *The Vienna Development Method: The Meta-Language.* Lecture Notes in Computer Science, Vol. 61. Springer.

[13] F. Boniol and V. Wiels. 2014. The Landing Gear System Case Study. In *ABZ (Case Study) (Communications in Computer and Information Science)*, Vol. 433. Springer, 1–18.

[14] P. Bourque and Richard E. Fairley (Eds.). 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0* (3rd ed.). IEEE Computer Society Press.

[15] M. Broy. 2018. A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability - from requirements to functional and architectural views. *Software and Systems Modeling* 17, 2 (2018), 365–393.

[16] J.M. Bruel, S. Ebersold, F. Galinier, A. Naumchev, M. Mazzara, and B. Meyer. 2020. Complementary materials on the role of formalism in software requirements. *CoRR* abs/1911.02564 (2020). arXiv:1911.02564

[17] Capterra. 2015. (2015). https://www.capterra.com/requirements-management-software/

[18] J.M. Carrillo-de-Gea, J. Nicolás, J.L. Fernández Alemán, J.A. Toval Álvarez, C. Ebert, and A. Vizcaíno. 2011. Requirements Engineering Tools. *IEEE Softw.* 28, 4 (2011), 86–91.

[19] N. Cataño and C. Rueda. 2009. Teaching Formal Methods for the Unconquered Territory. In *TFM (Lecture Notes in Computer Science)*, Vol. 5846. Springer, 2–19.

[20] F. E. Cellier. 1991. *Continuous system modeling*. Springer.

[21] P. Pin-Shan Chen. 2002. The Entity Relationship Model - Toward a Unified View of Data (Reprint). In *Software Pioneers*. Springer Berlin Heidelberg, 311–339.

[22] E. F. Codd. 1983. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM* 26, 1 (1983), 64–69.

[23] J.L. Colaço, B. Pagano, and M. Pouzet. 2005. A conservative extension of synchronous data-flow with state machines. In *EMSOFT*. ACM, 173–182.

[24] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard. 2007. Enhancing UML Extensions with Operational Semantics. In *MoDELS (Lecture Notes in Computer Science)*, Vol. 4735. Springer, 271–285.

[25] P. L. da R. Rodrigues, M. Ecar, S. V. Menezes, J. P. S. da Silva, G. T. A. Guedes, and E. de M. Rodrigues. 2018. Empirical Evaluation of Formal Method for Requirements Specification in Agile Approaches. In *SBSI*. ACM, 53:1–53:8.

[26] F. Dalpiaz. 2015. Teaching Goal Modeling in Undergraduate Education. In *iStarT@CAiSE (CEUR Workshop Proceedings)*, Vol. 1370. CEUR-WS.org, 1–6.

[27] A. Dardenne, A. van Lamsweerde, and S. Fickas. 1993. Goal-Directed Requirements Acquisition. *Sci. Comput. Program.* 20, 1-2 (1993), 3–50.

[28] N. Dean and M. Hinchey. 1996. *Teaching and Learning Formal Methods*. Academic Press.

[29] M. dos Santos Soares, J. L. M. Vrancken, and A. Verbraeck. 2011. User requirements modeling and analysis of software-intensive systems. *J. Syst. Softw.* 84, 2 (2011), 328–339.

[30] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. 1998. Property specification patterns for finite-state verification. In *FMSP*. ACM, 7–15.

[31] Event-b. 2018. (2018). http://www.event-b.org/

[32] M. Eysholdt and H. Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *SPLASH/OOP-SLA Companion*. ACM, 307–309.

[33] A. Ferrari, F. Dell'Orletta, A. Esuli, V. Gervasi, and S. Gnesi. 2017. Natural Language Requirements Processing: A 4D Vision. *IEEE Softw.* 34, 6 (2017), 28–35.

[34] A. Ferrari, G. Gori, B. Rosadini, I. Trotta, S. Bacherini, S. Fantechi, and S. Gnesi. 2018. Detecting requirements defects with NLP patterns: an industrial experience in the railway domain. *Emp. Soft. Eng.* 23, 6 (2018), 3684–3733.

[35] Eclipse foundation. 2015. Papyrus. (2015). https://eclipse.org/papyrus/

[36] R. B. France, S. Ghosh, T. T. Dinh-Trong, and A. Solberg. 2006. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer* 39, 2 (2006), 59–66.

[37] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. 1991. Informal and Formal Requirements Specification Languages: Bridging the Gap. *IEEE Trans. Software Eng.* 17, 5 (1991), 454–466.

[38] J. Gibbons and J. Nuno Oliveira (Eds.). 2009. *Teaching Formal Methods, Second International Conference, TFM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. Lecture Notes in Computer Science, Vol. 5846. Springer.

[39] J. P. Gibson and D. Méry. 1998. Teaching Formal Methods: Lessons to Learn. In *IWFM (Workshops in Computing)*. BCS.

[40] Mario Gleirscher, Simon Foster, and Jim Woodcock. 2019. New Opportunities for Integrated Formal Methods. *ACM Comput. Surv.* 52, 6, Article 117 (Oct. 2019), 36 pages.

[41] Mario Gleirscher and Diego Marmsoler. 2018. Formal Methods: Oversold? Underused? A Survey. *CoRR* abs/1812.08815 (2018). arXiv:1812.08815

[42] R. Gmehlich, K. Grau, F. Loesch, A. Iliasov, M. Jackson, and M. Mazzara. 2013. Towards a formalism-based toolkit for automotive applications. In *FormaliSE@ICSE*. IEEE Computer Society, 36–42.

[43] F.H. Golra, F. Dagnat, J. Souquières, I. Sayar, and S. Guérin. 2018. Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation. In *SEFM, LNCS*, Vol. 10886. Springer, 54–69.

[44] M. T. Grinder. 2002. Animating automata: a cross-platform program for teaching finite automata. In *SIGCSE*. ACM, 63–67.

[45] R. Hähnle, K. Johannisson, and A. Ranta. 2002. An Authoring Tool for Informal and Formal Requirements Specifications. In *FASE (Lecture Notes in Computer Science)*, Vol. 2306. Springer, 233–248.

[46] T. Hainey, T. M. Connolly, M. Stansfield, and E. A. Boyle. 2011. Evaluation of a game to teach requirements collection and analysis in software engineering at tertiary education level. *Comput. Educ.* 56, 1 (2011), 21–35.

[47] D. Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274.

[48] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. J. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan. 2009. Using formal specifications to support testing. *ACM Comput. Surv.* 41, 2 (2009), 9:1–9:76.

[49] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677.

[50] C. A. R. Hoare. 2002. An Axiomatic Basis for Computer Programming (Reprint). In *Software Pioneers*. Springer Berlin Heidelberg, 367–383.

[51] IBM. 2020. Rational Doors. (2020). https://www.ibm.com/products/requirements-management

[52] IBM. 2020. Rational Rhapsody 8.1.5. (2020). https://www.ibm.com/products/systems-design-rhapsody

[53] IEEE. 1998. *IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1998.* DOI: http://dx.doi.org/10.1109/IEEESTD.1998.88286

[54] IEEE. 2011. *ISO/IEC/IEEE Int. Standard - Systems and software engineering – Life cycle processes –Requirements engineering, ISO/IEC/IEEE 29148:2011(E), 2011.*

[55] M.H. L. Wong Cheng In. 1994. *Informal, semi-formal, and formal approaches to the specification of software requirements.* MSc. Dissertation. University of British Columbia.

[56] F. Ishikawa, K. Taguchi, N. Yoshioka, and S. Honiden. 2009. What Top-Level Software Engineers Tackle after Learning Formal Methods: Experiences from the Top SE Project. In *TFM*. Lec. Notes in Comp. Sc., Vol. 5846. Springer, 57–71.

[57] D. Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis.* MIT Press.

[58] M. Jackson. 2001. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley.

[59] B. Jeannet and F. Gaucher. 2015. Debugging real-time systems requirements: simulate the "what" before the "how". In *Embedded World Conf.* 6.

[60] M. Kassab, C. J. Neill, and P. A. Laplante. 2014. State of practice in requirements engineering: contemporary data. *Innov. Syst. Softw. Eng.* 10, 4 (2014), 235–241.

[61] L. Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley.

[62] P. Laplante. 2017. *Requirements Engineering for Software and Systems.* Auerbach Publications.

[63] P. Gorm Larsen, J. S. Fitzgerald, and S. Riddle. 2009. Practice-oriented courses in formal methods using VDM^{++}. *Formal Aspects Comput.* 21, 3 (2009), 245–257.

[64] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 1998. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*. 404–420.

[65] F.L. Li, J. Horkoff, A. Borgida, G. Guizzardi, L. Liu, and J. Mylopoulos. 2015. From Stakeholder Requirements to Formal Specifications Through Refinement. In *REFSQ (Lecture Notes in Computer Science)*, Vol. 9013. Springer, 164–180.

[66] S. Liu, K. Takahashi, T. Hayashi, and T. Nakayama. 2009. Teaching formal methods in the context of software engineering. *ACM SIGCSE Bull.* 41, 2 (2009), 17–23.

[67] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher. 2019. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Comput. Surv.* 52, 5 (2019), 100:1–100:41.

[68] R. R. Lutz. 1993. Analyzing software requirements errors in safety-critical, embedded systems. In *RE*. IEEE Computer Society, 126–133.

[69] J. Magee and J. Kramer. 2006. *Concurrency - state models and Java programs (2. ed.).* Wiley.

[70] A. Matoussi, F. Gervais, and R. Laleau. 2010. *An Event-B formalization of KAOS goal refinement patterns. [Research Report] TR-LACL-2010-1, LACL.* Technical Report. University Paris 12. 1–34 pages.

[71] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. 2009. Easy Approach to Requirements Syntax (EARS). In *RE*. IEEE Computer Society, 317–322.

[72] B. Meyer. 1985. On Formalism in Specifications. *IEEE Softw.* 2, 1 (1985), 6–26.

[73] B. Meyer. 1988, 1997. *Object-Oriented Software Construction (1st and 2nd editions).* Prentice-Hall.

[74] B. Meyer. 2003. A Framework for Proving Contract-Equipped Classes. In *Abstract State Machines (Lecture Notes in Computer Science)*, Vol. 2589. Springer, 108–125.

[75] B. Meyer. 2013. Multirequirements. *Modeling and Quality in Req. Engineering (Glinz Festscrhift)* (2013).

[76] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. 2009. Programs That Test Themselves. *Computer* 42, 9 (2009), 46–55.

[77] L. Mich. 1996. NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. *Nat. Lang. Eng.* 2, 2 (1996), 161–187.

[78] L. Mich, M. Franch, and P. Novi Inverardi. 2004. Market research for requirements analysis using linguistic tools. *Requir. Eng.* 9, 1 (2004), 40–56.

[79] R. Milner. 1980. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, Vol. 92. Springer.

[80] M. L. Minsky. 1967. *Computation: Finite and Infinite Machines.* Prentice-Hall.

[81] Modelica. 2017. (2017). https://doc.modelica.org/

[82] ModelioSoft. 2017. (2017). https://www.modeliosoft.com/products/modelio-sa-system-architects.html

[83] S. Moon, K. Hyung Lee, and D. Lee. 2004. Fuzzy branching temporal logic. *IEEE Trans. Syst. Man Cybern. Part B* 34, 2 (2004), 1045–1055.

[84] T. Nakatani. 2008. Requirements Engineering Education for Professional Engineers. In *JCKBSE (Frontiers in Artificial Intelligence and Applications)*, Vol. 180. IOS Press, 495–504.

[85] A. Naumchev. 2019. *Exigences orientées objets dans un cycle de vie continu.* Ph.D. Dissertation. Université de Toulouse, Université Toulouse III-Paul Sabatier.
[86] A. Naumchev. 2019. Object-Oriented Requirements: Reusable, Understandable, Verifiable. In *TOOLS, LNCS*, Vol. 11771. Springer, 150–162.
[87] A. Naumchev. 2019. Seamless Object-Oriented Requirements. In *2019 Int. Multi-Conf. on Engineering, Computer and Information Sciences (SIBIRCON).* 0743–0748.
[88] A. Naumchev and B. Meyer. 2016. Complete Contracts through Specification Drivers. In *TASE.* IEEE Computer Society, 160–167.
[89] A. Naumchev, B. Meyer, M. Mazzara, F. Galinier, J.M. Bruel, and S. Ebersold. 2019. AutoReq: Expressing and verifying requirements for control systems. *J. Comput. Lang.* 51 (2019), 131–142.
[90] A. Naumchev, B. Meyer, and V. Rivera. 2015. Unifying Requirements and Code: An Example. In *Ershov Memorial Conference (Lecture Notes in Computer Science)*, Vol. 9609. Springer, 233–244.
[91] T. Nguyen. 2015. Verification of Behavioural Requirements for Complex Systems with FORM-L, a MODELICA Extension. In *26th Int. Conf. on Software & Systems Engineering and their Applications.*
[92] OMG. 2011. Unified Modeling Language, Superstructure, Version 2.4.1. (2011). http://www.omg.org/spec/UML/2.4.1
[93] OMG, Sanford Friedenthal, Alan Moore, and Rick Steiner. 2008. SysMl Tutorial. (June 2008). http://www.omgsysml.org/INCOSE-2008-OMGSysML-Tutorial-Final-revb.pdf
[94] Object Management Group (OMG). 2007. OMG Systems Modeling Language (OMG SysML™), V1.0. (2007). OMG Document Number: formal/2007-09-01 Standard document URL: http://www.omg.org/spec/SysML/1.0/PDF.
[95] Object Management Group (OMG). 2015. UML 2.5. (March 2015). http://www.omg.org/spec/UML/2.5/
[96] S. Ouhbi, A. Idri, J. Luis Fernández Alemán, and A. Toval. 2015. Requirements engineering education: a systematic mapping study. *Requir. Eng.* 20, 2 (2015), 119–138.
[97] R. F. Paige and J. S. Ostroff. 2002. The Single Model Principle. *J. Object Technol.* 1, 5 (2002), 63–81.
[98] D. Lorge Parnas. 2010. Precise Documentation: The Key to Better Software. In *The Future of Software Engineering.* Springer, 125–148.
[99] J. L. Peterson. 1977. Petri Nets. *ACM Comput. Surv.* 9, 3 (1977), 223–252.
[100] P. Pettersson and K/ G. Larsen. 2000. Uppaal2k. In *Bull. Eur. Assoc. for Theoretical Computer Sc.*, Vol. 70. 40–44.
[101] The Overture Project. 2017. Overture Tool: Formal Modelling in VDM. (2017). http://overturetool.org/method/
[102] Protégé. 2016. (2016). https://protege.stanford.edu/
[103] Aarne Ranta. 2011. *Grammatical Framework - Programming with Multilingual Grammars.* Cambridge University Press.
[104] P. Raymond, Y. Roux, and E. Jahier. 2008. Specifying and Executing Reactive Scenarios With Lutin. *Electron. Notes Theor. Comput. Sci.* 203, 4 (2008), 19–34.
[105] J. N. Reed and J. E. Sinclair. 2004. Motivating Study of Formal Methods in the Classroom. In *TFM (Lecture Notes in Computer Science)*, Vol. 3294. Springer, 32–46.
[106] Respect-it. 2011. Objectiver V3. (2011). http://www.objectiver.com
[107] A. B. Romanovsky and M. Thomas (Eds.). 2013. *Industrial Deployment of System Engineering Methods.* Springer.
[108] A.W. Roscoe. 1997. *The Theory and Practice of Concurrency.* Prentice Hall.
[109] R. E. Schneider and D. M. Buede. 2000. Properties of a High Quality Informal Requirements Document. *INCOSE Springer Symp.* 10, 1 (July 2000), 352–359.
[110] S. A. Schneider, H. Treharne, and H. Wehrheim. 2014. The behavioural semantics of Event-B refinement. *Formal Aspects Comput.* 26, 2 (2014), 251–280.
[111] W. Scott and S. Cook. 2004. A Context-free Requirements Grammar to Facilitate Automatic Assessment. In *Ninth Australian Workshop on Requirements Engineering (AWRE 2004): Proceedings.* AWRE, 4.1–4.10.
[112] J. Slankas and L. Williams. 2013. Automated extraction of non-functional requirements in available documentation. In *2013 1st Int. Workshop on Natural Language Analysis in Software Engineering (NaturaLiSE).* IEEE Comp. Soc., 9–16.
[113] SPIN / Promela. 2020. (2020). http://spinroot.com/spin/Man/promela.html
[114] State Chart XML (SCXML) 2012. State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Working Draft 16 February 2012. (2012). https://www.w3.org/TR/scxml/
[115] Stimulus. 2015. (2015). http://argosim.com/product-overview/
[116] I. Stoica, R. Tappan Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. (2001), 149–160.
[117] W. Su and J.R. Abrial. 2017. Aircraft landing gear system: approaches with Event-B to the modeling of an industrial system. *Int. J. Softw. Tools Technol. Transf.* 19, 2 (2017), 141–166.
[118] Dassault Systems. 2016. CATIA Reqtify. (2016). https://www.3ds.com/products-services/catia/products/reqtify
[119] Sparx Systems. 2017. Enterprise Architect. (2017). https://sparxsystems.eu/enterprisearchitect/system-requirements/

[120] S. Tarkan and V. Sazawal. 2009. Chief Chefs of Z to Alloy: Using a Kitchen Example to Teach Alloy with Z. In *TFM (Lecture Notes in Computer Science)*, Vol. 5846. Springer, 72–91.

[121] Thomas Tilley, Richard Cole, Peter Becker, and Peter W. Eklund. 2005. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In *Formal Concept Analysis*. Lecture Notes in Computer Science, Vol. 3626. Springer, 250–271.

[122] N. Tillmann, F. Chen, and W. Schulte. 2006. Discovering Likely Method Specifications. In *ICFEM (Lecture Notes in Computer Science)*, Vol. 4260. Springer, 717–736.

[123] N. Tillmann and J. de Halleux. 2008. Pex-White Box Test Generation for .NET. In *TAP (Lecture Notes in Computer Science)*, Vol. 4966. Springer, 134–153.

[124] N. Tillmann and W. Schulte. 2005. Parameterized unit tests. In *ESEC/SIGSOFT FSE*. ACM, 253–262.

[125] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. 2015. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9035. Springer, 566–580.

[126] A. van Lamsweerde. 2000. Formal specification: a roadmap. In *ICSE - Future of SE Track*. ACM, 147–159.

[127] A. van Lamsweerde. 2001. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE*. IEEE Computer Society, 249.

[128] A. van Lamsweerde. 2004. Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice. In *Proc. Requirements Engineering Conf., 12th IEEE Int.* IEEE Computer Society, 4–8.

[129] M. von der Beeck. 1994. A Comparison of Statecharts Variants. In *FTRTFT (Lecture Notes in Computer Science)*, Vol. 863. Springer, 128–148.

[130] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.M. Bruel. 2009. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *RE*. IEEE Computer Society, 79–88.

[131] K. Wiegers and J. Beatty. 2013. *Software requirements*. Pearson Education.

[132] Jeannette M. Wing. 1988. A Study of 12 Specifications of the Library Problem. *IEEE Softw.* 5, 4 (1988), 66–76.

[133] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. 2009. Formal methods: Practice and experience. *ACM Comput. Surv.* 41, 4 (2009), 19:1–19:36.

[134] E. Yu. 1997. Towards modelling and reasoning support for early-phase requirements engineering. In *Proc. ISRE '97: 3rd IEEE Int. Symp. on Requirements Engineering*. IEEE Computer Society, 226–235.

[135] E. S. K. Yu and J. Mylopoulos. 1998. Why Goal-Oriented Requirements Engineering. In *REFSQ*. Presses Universitaires de Namur, 15–22.

[136] P. Zave. 2017. Reasoning About Identifier Spaces: How to Make Chord Correct. *IEEE Trans. Software Eng.* 43, 12 (2017), 1144–1156.

[137] P. Zave and M. Jackson. 1997. Four Dark Corners of Requirements Engineering. *ACM Trans. Softw. Eng. Methodol.* 6, 1 (1997), 1–30.

[138] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, M. A. Ajagbe, E.V. Chioasca, and R. T. Batista-Navarro. 2020. Natural Language Processing (NLP) for Requirements Engineering: A Systematic Mapping Study. *CoRR* abs/2004.01099 (2020).

[139] D. Zowghi and S. Paryani. 2003. Teaching Requirements Engineering through Role Playing: Lessons Learnt. In *RE*. IEEE Computer Society, 233.