Formality in software requirements

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

A major determinant of the quality of software systems is the quality of their requirements, which should be both understandable and precise. Natural language, the most commonly used for writing requirements, helps understandability, but lacks precision.

To achieve precision, researchers have for many years advocated the use of "formal" approaches to writing requirements. These efforts have produced many requirements methods and notations, which vary considerably in their style, scope and applicability. The present survey discusses some of the principal approaches.

The analysis uses a number of complementary criteria, such as traceability support, level of abstraction and tool support. It classifies the surveyed techniques into five categories: general-purpose, natural-language-based, graph and automata, other mathematical notations, and programming-language-based). The review includes examples from all of these categories, altogether 22 different methods, including for example SysML, Relax, Petri Nets, VDM, Eiffel, Event-B, Alloy.

The review discusses a number of important open questions, including 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, Alexandr Naumchev, Manuel Mazzara, and Bertrand Meyer. 2019. Formality in software requirements. 1, 1, Article 1 (November 2019), 50 pages.
DOI: 10.1145/nmnnnn.nmnnnn

Draft. Will be submitted to ACM Computing Surveys

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, fundamentally depend 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 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 the mathematical method and mathematical 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 to the field might assume that all software development starts with a formal description of the requirements. This approach is by far not the standard today; most practical projects, if they use explicit requirements at all, describe them informally, either in the form of a natural-language “requirements document” or (in agile methods) through individual “user stories”, also expressed in natural language. At best they will use a few isolated formal elements to cover some delicate, critical aspects, while the bulk of the description remains informal.

In contrast with this dominant practice, a number of *formal requirements methods* have been developed, and applied more or less widely. These formal methods, with the associated formal notations and supporting tools, are the subject of the present discussion and will be surveyed in detail in the following sections.

Before entering the core of the discussion, we need a few clarification on the terminology.

### 1.1 Requirements versus specifications

“Formal *specification*” is a well-accepted concept, covered by various survey articles [111, 124]. The present article surveys formal *requirements*. While it is generally accepted that the concepts of specification and requirements are both distinct and related, there exist – even within a single normative source such as the Software Engineering Body of Knowledge [18] – varying definitions of the difference. This article will rely on the following definition, with which we believe many knowledgeable professionals would agree.

First, what is common to requirements and specifications: both describe the “what” of a system or system element — its purpose and the constraints to which it is submitted — rather than the “how”, which is the responsibility of other software engineering tasks, *design* and *implementation*.

As to the difference, it is one of purpose and scope:

- A specification describes the *technical properties* of a system, or often some part of a system. For example, the rendering engine of a Web browser must display any given HTML text in a certain way. A specification will describe the desired properties of this rendering.
- Requirements describe the properties of a system or system element as relevant to its human users, or more generally its *environment*. (This generalization is necessary since not all systems have a direct human “user”: many, such as the engine of a car, serve the purpose of another system, in this case the car as a whole.) For example, the requirements for a Web browser describe the functionality that it provides to its users and the constraints on this functionality.

This definition indicates why the distinction between requirements and specifications cannot be absolute: to produce the requirements of a complex system, it will be necessary to decompose them into sub-requirements of its components, and the further you go into this decomposition the more detailed and technical the requirements will become, getting closer to specifications. Even if it is not absolute, however, the distinction is useful. This article focuses on formal approaches to requirements, applicable to entire systems and covering the properties directly relevant to the systems’ users and environment. Unlike formal specifications, the topic of formal requirements has not been the target of many systematic studies. The present article is an attempt to fill this gap.
1.2 Terminology note: “specification”

Two further observations on terminology will help avoid confusion:

- The verb “to specify” is in common English use to mean “to describe precisely” or “to include a mention of”. Such use is applicable to requirements too, as in “Your implementation ignores this case, but the requirements document specifies that it must be reported as an error!” It would be cumbersome to deprive ourselves from such standard usage, as long as it does not imply any confusion with the term “specification” in its technical sense discussed above.

- This article follows the practice, common in the requirements literature and standardized, of using “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 grammatical 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” etc.

2 CRITERIA FOR ASSESSING APPROACHES TO REQUIREMENTS

The matter of assessing the quality of requirements has received significant attention, not only in the research literature [102, 106] but also in industry standards, from the venerable IEEE 830-1993 [1] to the more recent ISO/IEC/IEEE 29148-2011 [2]. They list such criteria as traceability, verifiability, consistency, justifiability and completeness.

For the present discussion, we need criteria one notch higher in the abstraction scale, since the goal is not to evaluate the quality of requirements but to assess approaches to requirements engineering. The discussion retains nine dimensions of approaches to requirements, listed in table 1.

1. Audience
2. Level of abstraction
3. Associated method
4. Tool support
5. Traceability support
6. Non-functional requirements
7. Environment vs system
8. Verifiability
9. Semantic definition

Table 1. Properties of requirements

Criterion 1, Audience, addresses the level of expertise expected of people who will use the requirements. Do they need, for example, to have received formal methods training? Or are the requirements intended for use by any stakeholder?

Criterion 2, Level of Abstraction (abbreviated in the assessment sections as “Abstraction”), addresses the level of detail (of properties of the system under description) which the requirements may or must cover.

Criterion 3, Associated method (“Method”), assesses whether the approach includes a comprehensive methodology to guide the requirements process (as opposed to more method-neutral approaches, which provide requirements support but adapt to their users’ preferred methods).
Criterion 4, Tool Support (‘‘Tools’’), covers the availability of tools to support the approach, as opposed to approaches that are conceptual only.

Criterion 5, Traceability support (abbreviated as ‘‘Traceability’’), assesses how the approach handles one of the most important issues associated with requirements: traceability, one-way or back-and-forth, between elements of the requirements and their counterparts in the 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 (‘‘Coverage’’), addresses whether the approach covers only the description of functional properties of the system (the functions it must perform, and environment constraints on these functions) or extends to non-functional properties (affecting aspects other than the system’s function, such as performance and security).

Criterion 7, Environment vs System (‘‘Scope’’), refers to the classic Jackson-Zave distinction [128] between two complementary parts of the requirements: describing the environment (or ”domain”) in which the future system will operate, and the constraints it imposes, such as ”no car will travel faster than 250 km/hour” or ”any bank transfer above EUR 10,000 must be pre-authorized”; and the system (or ”machine”) which the project will build. Does the approach cover both, or will requirements describe only the system part?

Criterion 8, Verifiability, assesses whether an approach supports the possibility of formally verifying properties of the resulting requirements. For approaches that provide such facilities, ” formal verification” typically means mathematical proofs, preferably supported by tools since manual verification is not sufficient for large and complex systems.

Criterion 9, Semantic definition (‘‘Semantics’’), assesses the availability and scope of a precise (if possible, formal) definition of the approach.

3 RUNNING EXAMPLE

To illustrate and compare the approaches surveyed, it is useful to rely on a common example. A seminal survey that used such an example is Wing’s 1988 study of specifications of a simple library system [123]. To reflect the challenges of today’s demanding IT applications, we need a more difficult example. This article uses the Landing Gear System for airplanes (LGS), a case study [17] that has received wide attention in the requirements literature (including in the authors’ own previous work [78]). The LGS is a 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 that stores the gear in the retracted position, and a door attached to the box. The door and the gear are independently activated by a digital controller. The controller reacts to changes of position of a handle by initiating either gear extension or retraction process. In other words, the controller must align in time the events of changing the handle’s position and sending commands to the door and the gear actuators: doors are opened and closed and gears are either moving out (extension) or moving in (retraction). One may express these rules, still in natural language but more precisely, as follows (all applicable to ”command line” mode):

- (R11bis) If the landing gear command handle has been pushed down and stays down, then eventually the gears 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 gears 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 outgoing sequence is not observed.

We will see how to express these properties (a small subset of the entire LGS description) in the approaches surveyed.

4 CLASSIFYING APPROACHES TO REQUIREMENTS

There is a wide diversity of approaches to requirements. This section gives the criteria used to classify them, and introduces the approaches reviewed. The review itself is in the next section.

4.1 The classification

It will be convenient to classify them into five broad categories, listed in Table 2. The recapitulative comparison of approaches (Table 4 in section 6.2) will summarize their assessment against the given criteria.

|   | N. Natural language | S. Semi-formal | A. Automata, graphs | M. Mathematical notation (other than A) | P. Programming language |
|---|---------------------|----------------|---------------------|----------------------------------------|------------------------|

Table 2. Classification of requirements approaches

The criterion for the classification is the degree of formalism of the underlying notation:

• The natural language category includes approaches that express requirements in English or another human language, although they can restrict the degree of “naturalness” of the text.
• Semi-formal denotes, in this discussion, approaches that 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 an programming language (as in the last category).
• Automata, graphs covers approaches relying on notations based on automata or graph theory. They usually provide graphical support and may leave the mathematical basis implicit since graphs, in particular, can be used a diagrammatic tool without deep mathematical knowledge.
• Mathematical covers approaches relying on mathematical formalisms other than those of the previous category. The theoretical basis is, generally, mathematical logic.

4.2 Criteria for selecting approaches

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

• Widely used (as in the case of commercially available tools such as Doors).
• 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 two of the methods reviewed, Relax (5.1.2) and multirequirements (5.5).
4.3 List of approaches surveyed

Table 3 lists the approaches retained for this survey. The references to the corresponding publications also appear here. When these publications do not give an approach an explicit name, we devised one carrying the central concept, such as Requirements Grammar, and marked it with an asterisk*.

| Name               | Category                | References | Section |
|--------------------|-------------------------|------------|---------|
| Requirements Grammar* | Natural language       | [104]      | 5.1.1   |
| Relax              | Natural language       | [121]      | 5.1.2   |
| Stimulus           | Natural language       | [55]       | 5.1.3   |
| NL to OCL*         | Natural language       | [48]       | 5.1.4   |
| NL to STD*         | Natural language       | [7]        | 5.1.5   |
| NL to OWL*         | Natural language       | [58]       | 5.1.6   |
| Doors              | Semi-formal            | [49]       | 5.2.1   |
| Reqtify            | Semi-formal            | [108]      | 5.2.1   |
| KAOS               | Semi-formal            | [117]      | 5.2.2   |
| URN                | Semi-formal            | [10]       | 5.2.3   |
| SysML              | Semi-formal            | [83]       | 5.2.4   |
| URML               | Semi-formal            | [14, 46]   | 5.2.5   |
| Petri Nets         | Automata- or graph-based | [92]     | 5.3.1   |
| Statecharts        | Automata- or graph-based | [41]     | 5.3.2   |
| Problem Frames     | Automata- or graph-based | [54]     | 5.3.3   |
| FSP/LTSA           | Automata- or graph-based | [61]     | 5.3.4   |
| FORM-L             | Automata- or graph-based | [80]     | 5.3.5   |
| Event-B            | Mathematical notation  | [5]        | 5.4.1   |
| VDM                | Mathematical notation  | [15]       | 5.4.2   |
| Alloy              | Mathematical notation  | [53]       | 5.4.3   |
| Tabular Relations  | Programming language   | [90]       | 5.4.4   |
| Multirequirements  | Programming language   | [67, 77]   | 5.5     |

Table 3. Requirements approaches surveyed in this article

5 REVIEW OF IMPORTANT APPROACHES

We now explore the approaches listed, in the order of the five categories of the previous section, illustrating them through the example introduced in section 3 and evaluating them per the criteria of section 2.

5.1 Natural language

Natural language is, as noted, the dominant form of the requirements of practical projects in industry. A number of requirements approaches consequently start from natural language statements of requirements. Such approaches face a fundamental challenge: software construction needs a high degree of precision, but natural language is notoriously imprecise. ([64] is a detailed analysis of the problems of using natural language for requirements.) There are two ways of addressing these problems in practice:

- Analyze the natural-language requirements texts, performing Natural Language Processing (“NLP”) to extract precise information and in particular to detect possible inconsistencies. Examples of methods using this approach include NL to OCL, NL to STD and NL to OWL.
• *Constrain* the kind of natural language used in requirements to ensure some degree of precision, without going as far as the semi-formal and formal approaches studied next. Examples include Requirements Grammar (5.1.1), Relax (section 5.1.2) and Stimulus (section 5.1.3).

5.1.1 Requirements Grammar. Significant effort has been devoted to processing natural-language requirements automatically, with the purpose of detecting inconsistencies and, more generally, improving their quality; examples of such work include [69], from 1996, and, more recently, [105]. Full natural-language processing raises challenges at the frontier of artificial intelligence research. To make the task more tractable, the Requirements Grammar approach defines a structured subset of natural language through a context-free grammar. Then the idea is that requirements elicitation will lead to requirements in this language, making it possible to avoid 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 meeting, as in

```
if the gears are locked down, the doors shall be closed
```

Fig. 1 shows a representation of requirements R11bis (*When the command line is working, if the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gears will be locked down and the doors will be seen closed.*) in this approach. The grammar is not able to express this requirement as stated in section 3: each requirement can only involve one *Independent Clause*, with a single *Subject of interest*. In the given requirement, there are two subjects (the *gears* and the *doors*). A solution is to split this requirement into two:

- When the command line is working, if the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gears will be locked down.
- When the command line is working, if the gears are locked down, then eventually the doors will be seen closed.

Then we adapt the result to the grammar according to the following scheme, where keywords of the grammar appear in **boldface**:

**Requirement:** a sentence corresponding to a requirement.

**Independent Clause:** the mandatory part of the requirement, that describe 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:** the action of the constraint,
- **Adjective** or **Noun Phrase:** depending of the verb, a complement for the verb.

The result for the LGS example appears in Fig. 1, where the boxes show the hierarchical structure.

Assessing Requirements Grammar with respect to the criteria of section 2:

- **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 a bit contrived.
- **Abstraction:** this approach is for requirements only, not influenced by implementation concerns.
- **Method:** the approach does not assume a particular requirements engineering method.
If landing gear command handle stays pushed down, the gears shall be locked down. If the gears are locked down, the doors shall be closed.

Fig. 1. Representation of requirement R11bis with the canvas of the Scott and Cook Requirements Grammar

- **Tool**: the authors proposed a tool, named Badger\(^1\), to express requirements and analyze their lexical clauses.
- **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.
- **Scope**: the approach can cover all aspects of requirements, including both system and environment aspects.
- **Verification** is limited to the following properties: consistency (the lexical clause are analyzed to detect if all requirements follow the pattern).
- **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.

5.1.2 Relax. Relax [121] is a language for formal modeling of requirements of CAS (Complex Adaptive System). The Relax syntax is close to natural language. Fig. 2 shows requirements R11bis, R12bis, R21 and R22 expressed in it. Keywords such as AS EARLY AS POSSIBLE or AFTER express temporality of events. They are semantically defined through a fuzzy branching temporal logic (FBTL) [73], making it possible to submit the requirements to validation tools. For example, R11bis can be translated into FBTL as:

\[
AG(AX_{>(\text{landing\_gear\_command=down})_{d_1}}(AX_{\geq d_1}(AG(\text{gear = locked\_down}) \land AG(\text{doors = closed}))))
\]

(“In any state after the event ‘landing gear command as been locked down’, ‘gear is locked down and doors are closed’ becomes true in a near future state.”) The example illustrates support for “fuzzy” notions (the F in FTBL) such as “near”.

Relax places a particular emphasis on the expression of environment properties. One can express such properties directly through the keyword ENV, define “monitors” through MON, and express 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 the critical ones.

Assessing Relax with respect to the criteria of section 2:

- **Audience**: the notation uses natural language expressions, requirements are hence readable by any stakeholder.

\(^1\)The tool seems no longer to be available.
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 and stays 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 and stays up.

R21: The retraction sequence SHALL not be observed AS EARLY AS POSSIBLE AFTER the command handle remains in down position.

R22: The outgoing 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

- **Abstraction**: this approach is for requirements only, not influenced by implementation concerns.
- **Method**: the approach does not assume a particular requirements engineering method.
- **Tool**: only prototypes such as Xtext [32] editors have been developed around Relax.
- **Traceability**: the approach 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.
- **Scope**: the approach explicitly covers system and environment aspects.
- **Verification**: is linked to the fuzzy branching temporal logic capabilities.
- **Semantics**: the syntax of specifications is defined precisely (through a specific grammar) and the corresponding semantics is defined through FTBL (fuzzy branching temporal logic, as noted above).

5.1.3 *Stimulus*. The Argosim Stimulus\(^2\) tool expresses requirements in a natural-language-like syntax [55], similar to Relax. This language is directed at stakeholders involved in system development.

Fig. 3 shows an initial attempt at expressing requirements \(R11bis\) and \(R12bis\). This version actually applies to a more complete version of the LGS example, taking into account timing properties present in the original LGS paper but not included above: a 15-second duration for retraction and for the outgoing sequence.

The idea is that after writing such an initial version one may, by simulating the inputs and observing the outputs, detect possible 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 (requirement `[LS_RQ_001]`) uses both a `When` but also a `Do ... afterwards`. Without this clause, the `When` would mean that When the handle is down, within 15 seconds the doors shall be closed and the gears down, leaving the behavior undefined afterwards. With the `Do ... afterwards`, the meaning is that after the handle has been pushed down, within 15 seconds the doors shall be closed and the gears down, remaining so until a new event.

Such wrong behavior can be detected through the Stimulus model-checker, which simulates the system behavior by varying inputs. Users can observe the system reaction and correct the requirements as needed. Stimulus favors such a process of incremental improvement of the requirements.

Assessing Stimulus with respect to the criteria of section 2:

\(^2\)http://argosim.com/product-overview/
**Audience**: the language is close to natural language; requirements are 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.

**Tool**: Stimulus is both the name of the language and the name of the tool used to support this language.

**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.

**Scope**: the approach explicitly covers both system and environment aspects.

**Verification** is available through model-checking to simulate the system’s reaction to various inputs.

**Semantics**: The language is inspired by programming languages Lucid Synchrone [24] and Lutin [95].

### 5.1.4 NL to OCL

[48] introduced a syntax to express constrained natural language specifications. The tool is based on UML [85] and more specifically on OCL (Object Constraint Language) [84], a Design-by-Contract-like mechanism for expressing semantic constraints on systems modeled in UML. It formalizes constraints, originally expressed in constrained natural language, into OCL. The long-term goal is integration into the KeY Java-oriented formal verification project [9]. NL to OCL includes a tool for object-oriented modeling which should in the future allow non-experts to write constraints without recourse to formal methods.

**Operation** retract

**OCL**:

```
context LGS::retract()
pre: self.handle = up
post: self.doors = closed and self.gears = up
```

**English**: for the operation retract() of the class LGS, the following precondition should hold:

- the handle is up

and the following post-conditions should hold:

- the doors are closed
- the gears are up.

---

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

Fig. 4. Possible representation of requirement R12bis with the NL to OCL approach
Fig. 4 represents requirement R12bis with this approach. This example (not tested, since the tool is no longer available) shows how to match an English specification of the retract operation, including pre- and postcondition, with its representation in OCL.

Assessing this approach with respect to the criteria of section 2:

- **Audience**: the target users of OCL are software engineers able to read a UML diagram and possessing a basic understanding of logic and formal reasoning.
- **Method**: The implied methodology starts from constrained natural-language requirements and expresses them in UML, with OCL for expressing semantic properties.
- **Tool**: the original tool for translation to OCL [48] is no longer available. The UML part of the approach is supported by the wide range of available UML tools (including other approaches for verification such as [19]).
- **Traceability**: the requirements are expressed as contracts on operations, thus linked to specification. There is, however, no specific support for traceability.
- **Coverage**: the approach applies to requirements whose semantics can be expressed through contracts.
- **Scope**: the approach only covers system aspects.
- **Verification**: the idea is to use the KeY tool to verify consistency of requirements.
- **Semantics**: The OCL representation of requirements provides a semantic definition.

### 5.1.5 NL to STD

The methodology of [7] iteratively transforms natural-language requirements into State Transition Diagrams, defined as sets triples [initial state, transition, resulting state].

**Fig. 5. Transformation from NL requirements to STD**

Fig. 5 illustrates the transformation of requirements to transition diagrams. The first step {1} transforms requirements into diagram fragments. For example, the requirement R11bis yields a transition from an unknown state, when the handle is down, to a final state Gears locked down and doors closed. Such states will have to be refined later. Step {2} assembles these fragments into a single STD, representing the entire system but still not final since it may still contain unknown states. Step {2} adds missing information and repeats the process.

Assessing this approach with respect to the criteria of section 2:

- **Audience**: the requirements are first expressed in natural language, readable by all stakeholders. The transition diagrams resulting from the transformation in step 2 are meant for a
more expert audience, but still equipped with natural-language explanations coming from the original text.

- **Abstraction**: this approach is for requirements only, not influenced by implementation concerns.
- **Method**: the approach proposes a method as outlined (going from natural language to transition diagrams).
- **Tool**: The translation into transition diagrams is manual.
- **Traceability**: There is 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.
- **Scope**: the approach can cover both system and environment aspects.
- **Verification**: transition diagrams are formal texts, which can be submitted to tools.
- **Semantics**: state transition diagrams are a well-known notion with precise semantics.

### 5.1.6 NL to OWL

The approach of [58] translates natural-language requirements into an intermediate requirements modeling language that can be easily formalized in OWL [13].

In the example, requirement R11bis – *When the command line is working, if the landing gear command handle has been pushed down and stays down, then eventually the gears will be locked down and the doors will be seen closed* – yields two functional goals:

- $FG_{11-1} := \text{lock}\_\downarrow \langle \text{object: \{the gears\}} \rangle ; <\langle \text{trigger: push}\_\downarrow \langle \text{object: \{landing gear command handle\}} \rangle \rangle$
- $FG_{11-2} := \text{close}\langle \text{object: \{the doors\}} \rangle ; <\langle \text{trigger: FG}_{11-1}\rangle$

$FG_{11-1}$ states that the gears should be locked down a while after the landing gear command handle has been pushed down. $FG_{11-2}$ states the obligation to close the doors when the gears are locked down, as triggered by the first functional goal. R12bis can be modeled in a similar way.

The requirement 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 using only one functional goal:

- $FG_5 := \text{extend}\langle \text{object: \{the gears\}} \rangle ; <\langle \text{trigger: remains}\_\downarrow \langle \text{object: \{landing gear command handle\}} \rangle \rangle$

This functional goal models the need to do not observe retraction sequence – and so, extend the landing gears – when the handle remains down. Requirement R22 can be modeled in a similar way.

Here is the assessment per section 2 criteria.

- **Audience**: requirements engineers.
- **Abstraction**: requirements only, no tainting by implementation.
- **Method**: yes.
- **Tool**: none directly for the method; there are tools for OWL such as Protégé³.
- **Traceability**: requirements are decomposed into ontologies, sharing the same namespace, which can be used to create links between several requirements.
- **Coverage**: both functional properties (called functional goals) and non-functional ones (quality goals).
- **Scope**: both system and environment (represented through domain assumptions).
- **Verification**: no support for verification of requirements, although OWL does have a formal semantics.
- **Semantics**: from OWL.

³[https://protege.stanford.edu/](https://protege.stanford.edu/)
5.2 Semi-formal

A number of approaches use partially formalized notations. Among these there are both research tools and industrial products [35].

5.2.1 Doors and Reqtify.

Doors [49] and Reqtify [108] are widely used in industry. They are semi-formal in the sense that they require a partially structured approach to the management of requirements. While they are distinct products from different providers (respectively IBM Rational and Dassault Systems), they are often used jointly and we cover them together. In both cases the focus is not on producing requirements, but on managing requirements produced through diverse tools.

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 specific 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 other 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 the operational practice the requirements would be expressed in some document, e.g. Word or PDF. DOORS would then record this document and its various attributes (creation date, version number, priority...) in its database, supporting the management of these requirements throughout the project. Reqtify would support defining and managing a traceability between their various elements.

With respect to the criteria of section 2:

- **Audience**: Aimed at a large audience of stakeholders, no particular technical prerequisite.
- **Abstraction**: Focused on requirements no influence from implementation.
- **Method**: These are software tools, with no particular method attached.
- **Traceability**: Traceability is the strong point of Reqtify in particular, which offers support for tracing requirements from specification to design and code. For example, it makes it possible to import requirements expressed in a Microsoft Word document and link them to C code.
- **Coverage**: No particular restriction of coverage or scope
- **Semantics**: Nor formal semantics, no verification methodology.

5.2.2 KAOS.

KAOS [27], like i* ([125]), is based on the Goal-Oriented Requirements Engineering approach to requirements [117] [118] [126]. The key idea of this approach 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. (Specifically, R, A, D ⊨ G with sets of requirements R, domain assumptions A, domain properties D and goals G.)

Goals can be composite, expressed in terms of simpler goals through trees 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.
The LGS model of Fig. 6 covers entities including door, gear and handle. Both R11bis and R21 refine the goal «When handle is down, gears are extended and doors are closed», which assumes normal mode and is itself part of the refinement of a more global goal defining the safety of the whole system. «Outgoing operation» addresses R11bis and R21 by managing the LGS outgoing sequence: after execution of the LG command, the handle has been moved up, doors remain closed and gears locked down. Some agent, triggered by the event “handle pushed down” will be responsible for performing this operation.

Assessing this approach with respect to the criteria of section 2:

- **Audience**: KAOS requires modeling experience, and some training in the method.
- **Abstraction**: No influence from implementation. KAOS can in fact be described as particularly abstract since it focuses on a concept, goals, which is even higher than requirements.
- **Method**: KAOS includes a general methodology for modeling systems, specifying dependencies between requirements, and refining goals.
- **Tool**: Objectiver [97] supports the expression of user requirements and their refinement in KAOS.
- **Traceability**: There is support for linking to specification documents.
- **Coverage**: mostly functional requirements, but can include some non-functional ones.
- **Scope**: both system and environment through the notion of operational goal and expectation in the refinement process, as described above.
- **Verification**: No specific support (graphical notation).
- **Semantics**: behavioral goals can be formalized in a temporal logic such as LTL, or in event-B ([5]) [63].
5.2.3 **URN.** User Requirements Notation [10] is a recommendation of the International Telecommunication Union (standard Z.151, from 2008, updated 2012, third version in progress) for the modelling, analysis, specification and validation of requirements, used mainly for business process modeling.

The basic concepts are goals, scenarios, and links between such elements. URN combines two complementary views: static goals, through the Goal-oriented Requirement Language (GRL); and dynamic scenarios, through the Use Case Map (UCM) notation. Both notations support checking: for GRL models, through “strategies”, representing initial situations; for UCM models, through “scenarios”, similar to test cases. There is also a framework for formal verification of UCM ([42], including time extensions. For traceability:

- Internally: URN supports links connecting GRL and UCM models, enabling completeness and consistency analysis.
- With external notations: through such tools as jUCMNav ⁴, to integrate URN models with DOORS.

For verification, methodological elements support validating goal-oriented models and resolving conflicts ([43] [44]).

Figure Fig. 7 uses GRL to describe the goals of the LGS example. The overall goal *LGS Safe* is decomposed into two goals (among others), reflecting R21 and R22 and refined further into more specific goals reflecting R11bis and R12bis. A task contributes to a goal: the “outgoing” task contributes to R11bis and “retracting” to R12bis. Note that tasks may depend on resources (that is not the case here).

![Fig. 7. Partial URN diagram for LGS requirements R11 bis and R21 (jUCMNav)](https://www.openhub.net/p/jucmnav)

Assessing this approach with respect to the criteria of section 2:

- **Audience:** URN needs a specific training on GRL and UCM.
- **Abstraction:** The approach does not have implementation concerns
- **Method:** URN does not impose any development process, but tutorials about jUCMnav present methodological elements.

---

⁴https://www.openhub.net/p/jucmnav
• **Tools**: jUCMNav is the open-source Eclipse \(^5\) plugin that supports URN ([100]). Others include OpenOME \(^6\) Sandrila \(^7\) UniqueSoft \(^8\) ArchSync \(^9\) and TouchCORE \(^10\)

• **Traceability**: as noted, URN support links between model elements.

• **Coverage**: GRL focuses on requirements, especially non-functional ones. UCM is most useful for specifying functional requirements.

• **Scope**: URN is dedicated to systems and specifically to reactive systems and business systems.

• **Verification**: no formal support (beyond the methodological elements mentioned above).

• **Semantics**: from [11], “the URN standard describes the URN abstract and concrete syntaxes formally, together with well-formedness constraints. However, the semantics is currently described more informally”.

### 5.2.4 SysML (Systems Modeling Language)

SYStem Modelling Language \([83]\) is an extension of UML \([85]\) dedicated to systems engineering. SysML provides requirements diagrams, requirements diagram of SysML allows users to express requirements in a textual representation, and covers non-functional requirements. The requirements diagram of SysML provides ways to express traceability links between different requirements (containment, derive, copy, trace) or between requirements and implementation elements (satisfy, verify, refine). SysML supports the expression of relationships between requirements and other modeling artifacts (like blocks, use cases, activities, . . .), allowing some verifications (e.g. each requirement is supported by at least one modeling element, . . .). But a formal expression of requirements and thus their formal verification is not possible.

The SysML requirement diagram of Fig. 8 expresses the imposition of LGS requirements R11bis to R22 on the landing set and its operations, as described in a block definition diagram, cf. table Fig. 9).

Assessing this approach with respect to the criteria of section 2:

• **Audience**: SysML is a modeling language needing specific knowledge.

• **Abstraction**: SysML stands at a high level of abstraction. Its requirement diagram enables quick requirements analysis and visual design.

• **Method**: SysML, like UML, is a notation, providing no methodology.

• **Tool**: It is supported by a number of tools such as IBM Rhapsody \([50]\), Modelio \([72]\), Enterprise Architect \([109]\), Papyrus \([33]\), . . . that implement a methodology preconised for using SysML.

• **Traceability**: SysML provides traceability links

• **Coverage**: SysML supports both functional and non-functional requirements.

• **Scope**: SysML focus on systems’ requirements and more particularly complex systems’ requirements.

• **Verification**: SysML does not allow any verification

• **Semantics**: SysML does not provide any semantic definition of the approach.

### 5.2.5 URML

User Requirements Modeling Language \([14, 46]\) is a UML profile developed in collaboration between Technische Universität München and Siemens Corporate Research. URML is

---

\[^{5}\]http://www.eclipse.org

\[^{6}\]https://se.cs.toronto.edu/trac/ome

\[^{7}\]http://www.paulherber.co.uk/visio-sdl/index.php

\[^{8}\]http://www.uniquesoft.com

\[^{9}\]https://sourceforge.net/projects/archsync/

\[^{10}\]http://touchcore.cs.mcgill.ca
Formality in software requirements

Fig. 8. Excerpt of the SysML functional requirements diagram of LGS (Papyrus)

Fig. 9. Excerpt of the description of LGS' requirements (Papyrus)

a language in its own, right based on MOF. However the meta-model of URML has been mapped to a UML profile in order to implement the CASE Tools supporting the approach. So, URML provides a graphical way to express requirements and notions linked to these requirements (e.g., threats, hazards and mitigations or even product lines and stakeholders). URML is an icon-based notation that also proposes a concrete syntax with a notational-only element. Hence, some semantics can be put on the links between requirements and other artifacts (presupposes, details, constrains, refines, with, . . .). It is focused on systems, but also supports the modeling of the environment (or context) of the system under analysis. URML is more particularly dedicated to requirements elicitation, and does not provide any guideline nor methodology to construct models, but it can support any methodology whose main concepts can be mapped to parts of its meta-model. Actually, it unifies concepts from goal-oriented, feature-oriented, process-oriented, and risk-oriented approaches and integrates them into models with functional and nonfunctional requirements.

Fig. 10 is an excerpt of a URML model of the LGS. To ensure a basic functionality of the LGS and its safety, we need to consider the outgoing sequence of the gears. This process is supposed to realize a goal that is «when the LG command is down, gears are extended and doors are closed». To do so, it requires both R21 and R11bis. R11bis is considered as a refinement of R21, since it expresses more precisely the same need.
Assessing this approach with respect to the criteria of section 2:

- **Audience**: As URML is an icon-based graphical modeling language, the learning process of abstractions of this requirements language will be easy.
- **Abstraction**: URML is based on abstractions of requirements.
- **Method**: This approach does not assume a particular requirements engineering method.
- **Tool**: It is supported by an Enterprise Architect Add-On. [109].
- **Traceability**: The links between concepts can lead to traceability.
- **Coverage**: URML integrates functional and nonfunctional requirements.
- **Scope**: It is focused on systems and particularly complex systems.
- **Verification**: No mean is provided to do verification of requirements.
- **Semantics**: Some semantics can be put on links between requirements and other artifacts.

### 5.3 Graph and automata

A number of approaches have a graph/automata basis. This mathematical object is well known in computer science in general [71], and also very often used to graphically capture behaviors expression. An automaton is composed of states, which represent how a program should react to a particular input (or event) and optionally end in another state (through a transition). An automaton is an abstract machine that runs automatically, reacting to events, until it reaches a state where, either there is no more inputs or it reaches an ending state (see for example Fig. 12 for a very abstract illustration).

This is why those approaches are used, most of the time, to represent the dynamic aspects of a system (behavior, timing constraints, etc.). In this survey we will focus in particular on Petri Nets [92], Statecharts [41], Problem Frames [54], FSP/LTSA [61], and FORM-L [80]. There are also several other approaches (UML Activity Diagrams and State Diagrams [81], UPPAAL [93], DEVS [21], SCXML [3] . . . ), but they can be considered as flavors and variations of the main one below.
5.3.1 **Petri Nets.** [92] are directed graphs made of nodes (representing transitions by bars) and places (representing conditions by circles) linked by arcs (representing transitions), as illustrated in Fig. 11.

They have been used in numerous application areas in the past. In [130] a good analysis is presented showing pros and cons of the approach, existing applications and limitations (such as lack of compositionality).

![Fig. 11. Petri Net model of an Electrical Circuit of Negative Pressuring Electro-valve (taken from [119])](image)

Assessing Petri nets with respect to the criteria of section 2:

- **Audience:** the notation, while graphical, is very specific and is targeted to Petri nets specialists only.
- **Abstraction:** this approach is for requirements only, not influenced by implementation concerns, and is very abstract due to its specific notation.
- **Method:** the approach does not assume a particular requirements engineering method.
- **Tool:** Petri nets is an old notation and lots of tools have been developed to edit, execute and verify them.
- **Traceability:** the approach focuses on the expression of dynamic behavior requirements, and offers no specific support for traceability.
- **Coverage:** Petri nets mainly address functional requirements.
- **Scope:** Petri nets only cover, mainly graphically, the modeling of the precise behavior of some parts of the system, without particular focus on the modeling of the environment nor the systems’ architecture. Is is hence limited to fully express requirements of a system.
- **Verification:** is very well supported by a lot of available tools around Petri nets. The verification tools are mainly based on model checking.
- **Semantics:** Petri nets have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis.

5.3.2 **Statecharts and state diagrams.** State charts [41] is a kind of state diagram used to describe the behavior of systems. A system is described through a finite number of states and transitions between those states (see Fig. 12).

Assessing State diagrams with respect to the criteria of section 2 has lots of commonalities with Petri Nets:
Fig. 12. State transition representation of the Gear System doors’ behavior, from closed to open state (taken from [16])

- **Audience**: the notation, while graphical, is very specific and is targeted to State diagrams specialists only.
- **Abstraction**: this approach is for requirements only, not influenced by implementation concerns, and is very abstract due to its specific notation.
- **Method**: even if there are some timing flavors of state diagrams, non-functional properties are not trivial to express with such formalisms.
- **Tool**: many tools and language (e.g., UML) have adopted state diagrams so they are highly tool supported. The counterpart of this is the number of different semantics that is around.
- **Traceability**: the approach focuses on the expression of dynamic behavior requirements, and offers no specific support for traceability.
- **Coverage**: State diagrams mainly address functional requirements.
- **Scope**: Similarly to Petri Nets, there is no specific modeling of the environment.
- **Verification**: is very well supported by a lot of available tools around State diagrams. The verification tools are mainly based on model checking.
- **Semantics**: due to the variety of state diagrams flavors, there is no unique semantics.

### 5.3.3 Problem Frames

[54] is an approach to software requirements analysis developed by Michael A. Jackson in the 1990s. It emphasizes on the relationships of the system in its real world environment.

Its interest relies also in the way requirements are decomposed. It is strongly related to M. Jackson’s methodology but has no particular traceability nor non-functional requirements support. The approach is supported by a set of tools, some of them graphical such as context diagrams or problem diagrams (e.g., Fig. 13).

Fig. 13. Generalized Problem Diagram

Assessing Problem Frames with respect to the criteria of section 2:
- **Audience**: the notation, while intuitive, is very specific and is targeted to Problem Frames specialists only.
• **Abstraction:** this approach is for requirements only, not influenced by implementation concerns, and is very abstract due to its specific notation.

• **Method:** the approach is strongly related to M. Jackson’s methodology but has no particular traceability nor non-functional requirements support.

• **Tool:** Problem Frames is a conceptual approach, not supported by any particular tool.

• **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.

• **Scope:** Problem Frames particularly focus on the modeling of the system in its environment, making even the distinction between the real environment, and the environment as perceived by the system itself.

• **Verification:** the approach does not convey any verification purpose.

• **Semantics:** While precise, the graphical notation of Problem Frames has no precise semantics.

5.3.4 **FSP/LTSA.** While Model-based formalisms such as Event-B and VDM are concerned with functional properties and sequential behavior, FSP (Finite State Processes) is a process algebra. Process algebras are concerned with interaction between concurrent processes. Among the original methods in this field, we can mention CSP [47] and CCS [70]. FSP inherits from both of them.

Mobile process algebras (e.g. Milner’s $\pi$-calculus [70]) represent a further development by addressing channel mobility. The basic unit of all process algebras is a communicating concurrent process with its own input and output ports, which makes feasible modeling the interaction with the environment and expressing requirements beyond the system itself. FSP not only offers mechanisms to describe processes’ behaviors, but also supports expression and verification of liveness, safety, progress, fairness and temporal properties through the powerful tool that supports FSP — LTSA.

LTSA (Labeled Transition System Analyzer) [61] is a tool for specification, verification, and animation of FSP models. Given an FSP model, the LTSA tool generates an executable Labeled Transition System (LTS) suitable for automated analysis and animation. A key property of LTSA is compositionality: systems are specified and modeled in separate parts or views with the interactions and synchronizations as shared actions or events. These can then be composed to form overall specifications or models using parallel or sequential composition, and model checked for satisfaction of safety and liveness properties. In our FSP interpretation of the LGS example, we exploit the compositionality in the following way:

$$ || \text{LGS} = (\text{LGS\_BEHAVIOR} || \text{CONTROL\_HANDLE}) $$

The “LGS” process is the parallel composition of the “LGS\_BEHAVIOR” process and the “CONTROL\_HANDLE” process. The “LGS\_BEHAVIOR” process specifies the LGS’ reaction to the changing state of the handle. The “CONTROL\_HANDLE” process specifies how the state of the handle may be changed. The two processes have intersecting alphabets and interact by means of the shared transitions (“up” and “down”) constituting the intersection.

FSP models are event-based specifications: the task of specifying an FSP model amounts to specifying all possible event-based traces of the process. The LGS requirements, however, are state-based, which one can see from such conditions as “if the landing gear command handle has been pushed up and stays up”. What does the “stays up” mean? FSP’s rely on so called *fluents* to specify states. To specify a fluent means to name the actions that set the fluent, and then name the actions that unset it. The “handle has been pushed up and stays up” condition from requirements “R12bis” and “R22” becomes the following fluent:

```
fluent HANDLE\_IS\_UP = \{up\}, \{down\}
```
The “up” transition sets the “HANDLE_IS_UP” fluent, and the “down” transition unsets it; no other transitions can set or unset the fluent. The “HANDLE_IS_DOWN” fluent for requirements “R11bis” and “R21” looks similarly:

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

One can safely treat pushing the handle down and pulling it up as instantaneous transitions. For transitions that have duration, specification of fluents becomes trickier. In our FSP interpretation of the LGS example, we treat closing the door, extending the gear, and retracting the gear as transitions that have durations and may be canceled in the process by the handle taken to another position. Because FSP transitions are instantaneous, we introduce several auxiliary FSP transitions that reflect the non-instantaneous real-world transitions: “start_closing”, “end_closing”, “start_extension”, “end_extension”, “start_retraction”, “end_retraction”. The following fluents build on these auxiliary FSP transitions:

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 fluents have several FSP transitions that unset them. The “start_retraction” transition, for example, sets the “GEAR_IS_RETRACTING” fluent, while transitions “end_retraction” and “start_extension” unset it; no other transitions can set or unset the fluent. With these fluents, we express the requirements from the running example through the following assertions:

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))

In these assertions “[]” stands for the “always” LTL operator, “⟨⟩” stands for the temporal “eventually”, “!” stands for the logical negation, “&&” stands for the logical conjunction.

LTSA, by default, assigns equal priorities to all transitions in an FSP model to guarantee fairness. Formally, property “always eventually a” holds for each transition “a” from the alphabet of an FSP. This makes the antecedents in the “R21” and “R22” assertions contradictory: under no circumstances can “[] HANDLE_IS_DOWN” become true because action “up” will always eventually happen and thus invalidate the “HANDLE_IS_DOWN” fluent. To handle such situations, FSP’s offer the lower priority operator. To make it possible for the “[] HANDLE_IS_DOWN” property to eventually hold, we downgrade the default priority of the “up” transition through the lower priority operator denoted in FSP’s with the “⟩⟩” notation:

||LGS = (LGS_OPTIONS || CONTROL_HANDLE) >> {up}.

The following auxiliary assertion checks that the lower priority operator has correctly done its job:

assert EVENTUALLY_ALWAYS_DOWN = <> [] HANDLE_IS_DOWN

After the auxiliary assertion passes the LTSA checks, it is safe to check assertions “R21” and “R11bis”. To check assertions “R22” and “R12bis”, one needs to downgrade the default priority of the “down” transition:

||LGS = (LGS_OPTIONS || CONTROL_HANDLE) >> {down}.

and then check that the handle eventually goes to the “up” position forever:

assert EVENTUALLY_ALWAYS_UP = <> [] HANDLE_IS_UP
Appendix A contains the complete FSP model for the LGS example. An interested reader may put it into the LTSA analyzer as it is to do some experimenting.

Assessing LTSA with respect to the criteria of section 2:

- **Audience**: the FSP notation is mathematically formal and requires the corresponding qualification both from the specifiers and the readers.
- **Abstraction**: the FSP abstraction is suitable both for specifying systems’ implementations and for specifying requirements.
- **Method**: the approach does not assume a particular requirements engineering method.
- **Tool**: the LTSA tool supports model checking FSP specifications, execution of specifications, and graphical simulation.
- **Traceability**: the approach focuses on model checking FSP specifications, so it offers no specific support for traceability.
- **Coverage**: LTSA allows for specifying and model checking liveness, safety, progress, and fairness properties.
- **Scope**: the approach can cover all aspects of requirements, including both system and environment aspects.
- **Verification**: the approach allows for verification of liveness, safety, progress, and fairness properties.
- **Semantics**: the FSP semantics is rigorously defined [61].

5.3.5 **FORM-L**. FORM-L [80] is a language extension aiming at formally modeling requirements and assumptions in MODELICA (see Fig. 14). While MODELICA is focusing mainly on the system itself, FORM-L allows both a formal modeling of the behavioral requirements that must be satisfied by a system, but also any assumptions made regarding the system’s environment. FORM-L was the result of the ITEA2 MODRIO project which goal was to improve MODELICA’s ability to support key systems engineering activities. It only addresses the early stage of the system development, but with a level of details in the requirements that allows some early validation as soon as a first, high-level, model of the system is available. There is no particular method attached to FORM-L, neither any support for traceability nor support for non-functional requirements. Thanks to its formal semantics and to the support of a transformation (e.g., to Stimulus), the FORM-L requirements can be verified by model-checking.

Assessing FORM-L with respect to the criteria of section 2:

- **Audience**: the FORM-L notation is mathematically formal and requires the corresponding qualification both from the specifiers and the readers.

\[ \text{requirement r11 is} \]
\[ \text{after (lgsHandle becomes down)} \]
\[ \quad \text{and (not failure)} \]
\[ \quad \text{and not (allGearsDown and allDoorsClosed)} \]
\[ \text{within 15s} \]
\[ \text{check (allGearsDown and allDoorsClosed) becomes true} \]
\[ \text{or lgsHandle becomes up} \]
\[ \text{or failure becomes true;} \]

Fig. 14. Example of FORM-L requirement

\[^{11}\text{MOdel DRIven physical systems Operation} \]
\[^{12}\text{http://argosim.com/}.\]
• **Abstraction**: the FORM-L abstraction is suitable both for specifying systems’ detailed design and for specifying goals and requirements.
• **Method**: the approach does not assume a particular requirements engineering method, but the main definition steps focus on: Goals, Requirements, Specification, Design.
• **Tool**: the FORM-L notation has only internal EDF tool support for code generation allowing simulation.
• **Traceability**: the approach aims at supporting traceability but has no support mechanism for now.
• **Scope**: the approach can cover all aspects of requirements, including both system and environment aspects.
• **Verification**: the approach allows for simulation.
• **Semantics**: the FORM-L notation is in the process of being formalized.

5.4 Other Mathematical

Here we will discuss other approaches to software requirements that have a mathematical basis other than graph and automata. In particular we will focus on Event-B, Vienna Development Method (VDM), FSP/LTSA, Alloy and Tabular Relations.

5.4.1 Event-B. Event-B [5] is a formal method for system-level modeling and analysis, consisting of modeling the system’s state in terms of sets and functions, and modeling state transformation using events. Event-B uses set theory (as a modeling notation in association with a formal semantic) and refinement (to represent systems at different abstraction levels). Mathematical proof is used to verify consistency between refinement levels.

To illustrate the general idea of Event-B we show model fragments of the LGS. Fig. 15 shows an event modeling the extension of the gear. Here a Boolean variable `gear_extended_p` is introduced to formalize whether a gear is extended or not.

![Make_GearExtended](image)

**Fig. 15.** Event to model gear extension (edited from [62])

Fig. 16 shows an event modeling the retraction of the gear. A second Boolean variable `gear_retracted_p` is here introduced to model if the gear is retracted or not.

Requirements can be expressed under the form of invariants. Fig. 17 shows an invariant stating that the a gear cannot be extended and retracted at the same time.

Assessing Event-B with respect to the criteria of section 2:

• **Audience**: The notation, based on basic math, is intended for specialists and software engineers with some formal background, while it could be hard to be used by stakeholders with no knowledge of formal approaches.
• **Abstraction**: The fundamental abstraction is the event, which describes the operational reaction to environment events.
Method: Event-B is a method based on refinement, not just a notation. The approach does not assume a particular requirements engineering method, though.

Tool: Tools exist for many years (for example Rodin\(^{13}\)).

Traceability: Traceability of requirements is not natively supported. Requirements could be partially traced from abstract to concrete, following the specific identifiers used through the process; however this approach is not scalable to any real-size system and traceability is not supported by tools.

Coverage: Both requirements regarding the system and the surrounding environment can be represented and modeled within this framework.

Scope: The system, however, is not effective to describe non-functional requirements.

Verification: Requirements expressed in Event-B are verifiable since the proof-based refinement process leading to implementation ensures that whatever described at the highest level of abstraction is preserved in the effective system. Event-B has been used with success in several industrial projects, for example in the transportation and aerospace field, as well as in business management [98]. In combination with other requirements approaches, such as Problem frames [54], it has also been used in automotive industry [38].

Semantics: the approach is operational, and via the stepwise refinement method more and more operational details are added to the model as long as the designer proceed in following the methodological approach. The idea is to create a chain of models from abstract to implementation, from which code could be, in principle, generated. The behavioural semantics of Event-B refinement has been described in [103].

5.4.2 Vienna Development Method. The Vienna Development Method (VDM) [15] is considered the first formal method of history for the development of computer systems. It was originally developed at the IBM Laboratory in Vienna in the 1970s. It includes the VDM Specification Language (VDM-SL) [94] and its extended form (VDM++) [31]. VDM-SL uses modules, whilst VDM++ (being object-oriented) uses classes with multiple inheritance. Computing systems may be modeled in VDM-SL at a higher level of abstraction than is achievable using programming languages and

\(^{13}\)http://www.event-b.org/
models can be transformed into detailed system designs through a refinement process (reification), that represents the method associated to the notation, in the same way it happens with Event-B. For the same reasons, problems with traceability are the same as with Event-B.

VDM-SL allows users to express different states of the systems and invariants that shall be met. For example, requirements R21 and R22 are expressed as invariants (inv) in List. 1.

```vdm-slip
state LGS of
gears: <retracted> | <extended> | <retracting> | <extending>
doors: <opened> | <closed> | <opening> | <closing>
handle: <down> | <up>

inv mk_LGS(gears, doors, handle) ==
(handle = <down> => gears <> <retracted>) and
(handle = <up> => gears <> <extended>)

init lgs == lgs = mk_LGS(<extended>, <closed>, <down>)
end
```

Listing 1. VDM-SL representation of the LGS states, with requirements R21 and R22

Another way to introduce requirements into a VDM specification is to translate the requirements as pre- and post-conditions of operations. For example, the requirement R11bis (resp. R12bis) is a description of what happens when handle remains down (resp. up). These operations are described List. 2, the pre and post conditions expressing the condition to call the operation and the state that should be reached after execution.

```vdm-slip
operations
extension_sequence()
ext wr gears
wr doors
rd handle
pre handle = <down>
post handle = <down> => (gears = <extended> and doors = <closed>);

retraction_sequence()
ext wr gears
wr doors
rd handle
pre handle = <up>
post handle = <up> => (gears = <retracted> and doors = <closed>);
end LGS
```

Listing 2. VDM-SL operations specified by requirements R11bis and R12bis

Moreover, keywords rd (for read-only) and wr (for read-write) ensure that only the state of the doors and the gears can be modified by the operations.

Assessing VDM with respect to the criteria of section 2:

- **Audience**: As a model-based formalism, VDM requires an understanding of some mathematical foundations, being not entirely suitable to customers or stakeholders with limited experience in these matters.
- **Abstraction**: VDM is based on mathematical foundations and hence is used at an early stage of the development process.
• **Method:** VDM supports the description of data and functionality. Data are defined by means of types. Functionality is defined in terms of operations.

• **Tool:** VDM is mature and has an extensive tool support (e.g., Overture).

• **Traceability:** Traceability is not supported.

• **Coverage:** Requirements regarding both the system and the surrounding environment can be modeled.

• **Scope:** VDM is not suitable to describe non-functional requirements.

• **Verification:** Requirements expressed in VDM are verifiable to the same extent they are verifiable in Event-B.

• **Semantics:** VDM has a formal ISO Standard semantics.

5.4.3 **Alloy.** Alloy [53] is a declarative modeling language based on first-order logic for expressing complex behavior of software systems. It is a subset of Z [6], sharing therefore the level of abstraction and the intended audience: generally specialists with knowledge of math (even if Alloy’s syntax was influenced by modeling languages like UML [101]). Alloy has formal syntax and semantics and brings to Z-style specification the automation offered by model checkers. Models can indeed be automatically checked for correctness using the Alloy Analyzer, a constraint solver that provides fully automatic simulation and checking based on a back-end SAT framework. Alloy does not come with an associated method like Event-B, and it is not intended for the description of non-functional properties, such as usability, performance, size, and reliability or intended for the description of timed behavior. Traceability of requirements is not natively supported.

Alloy is a younger modeling languages with respect to others presented in this section. However, the research community is active on it and applications have already shown the validity of the approach. For example, in [127] Alloy is used for formal modeling of the Chord ring-maintenance protocol [107]. This paper provides the first specification of correct operations and initialization for Chord, and uses a formal model in Alloy to provide a proof of correctness.

Alloy features no native mechanism for expressing properties of the form “handle remains in the DOWN position”, like the ones used in requirements “R21” and “R22” from the running example. In Alloy it is possible, however, to characterize state transitions. The formulations of the requirements “R21” and “R22” may be relaxed without relaxing their semantics:

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

**R22** If the landing gear command handle is up, the gear is not extending.

We do not need the “remains” condition: if some property holds in an arbitrary state under an assumption, the same property will hold in every other state under the same assumption. Alloy is capable of capturing such relaxed formulations:

```plaintext
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

R22: check { all lgs, lgs': LGS |((lgs.handle in Up) and (lgs'.handle in Up) and Main [lgs, lgs']) implies (lgs'.gear not in Extending) } 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 “5” number represents the size of the search space: Alloy relies on counterexample-based bounded model checking.

http://overturetool.org/method/
Due to the absence of temporal operators, we also refine requirements “R11bis” and “R12bis” to express them in Alloy. The refined versions are:

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

**R12bis** If the handle remains up, three transitions of the LGS will suffice to have the gear retracted and the door closed.

The following Alloy assertions capture the modified requirements:

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

R12bis: check { all lgs1, lgs2, lgs3, lgs4 : LGS | {{(lgs1.handle in Up and lgs2.handle in Up and lgs3.handle in Up and lgs4.handle in Up) and (Main [lgs1, lgs2] and Main [lgs2, lgs3] and Main [lgs3, lgs4])}} implies (lgs4.gear in Retracted and lgs4.door in Closed)) } for 5

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

Assessing Alloy with respect to the criteria of section 2:

- **Audience**: the Alloy notation is mathematically formal and requires the corresponding qualification both from the specifiers and the readers.
- **Abstraction**: the Alloy abstraction is suitable both for specifying systems’ implementations and for specifying requirements.
- **Method**: the approach does not assume a particular requirements engineering method.
- **Tool**: the Alloy analyzer tool supports model checking Alloy specifications, execution of specifications, and graphical simulation.
- **Traceability**: the approach focuses on model checking Alloy specifications, so it offers no specific support for traceability.
- **Coverage**: Alloy allows for specifying and model checking behavioral specifications, without consideration of non-functional properties.
- **Scope**: the approach can cover all aspects of requirements, including both system and environment aspects.
- **Verification**: the approach allows for counterexample-guided model checking of models’ consistency and bounded model checking of first-order predicate logic assertions over these models.
- **Semantics**: the Alloy semantics is rigorously defined in [53].

Appendix B includes the complete Alloy example that we have specified and checked.

### 5.4.4 Tabular relations.

David Lorge Parnas in his seminal work [88] proposes to represent relations over a system’s variables in the tabular form. He assumes the general case of \( n \)-dimensional tables with \( (n - 1) \)-dimensional headers. The tabular notation is mathematically precise. The author assumes the specifier to be familiar with predicate logic and to have substantial abstraction skills. The mathematical preciseness of the approach makes it a subject to formal analysis, verification, and simplifies development of tool support. We found, however, no tools supporting the approach. Development of such tools might be a perspective direction for further development of the approach.
Fig. 18. Expressing requirements “$R_{11\,bis}$” and “$R_{12\,bis}$” with a vector function table [88]. The “NC(handle_position)” predicate states that the value of variable “handle_position” does not change [89].

| gear_status' | extended | retracted |
|-------------|----------|-----------|
| door_status' | closed   | closed    |

| handle_position = down ∧ NC(handle_position) | handle_position = up ∧ NC(handle_position) |
|---------------------------------------------|---------------------------------------------|
| gear_status | gear_status ≠ retracting | gear_status ≠ extending |

Fig. 19. Expressing requirements “$R_{21}$” and “$R_{22}$” with a vector relation table [88]. The table describes the relation connecting the handle’s position and the gear’s status.

Tabular representations do not support temporal operators. To capture requirements “$R_{11\,bis}$” and “$R_{12\,bis}$”, we introduce variables “gear_status’” and “door_status’”. These variables denote the values of variables “gear_status” and “door_status”, respectively, when the program finishes its execution. The horizontal header of the resulting table (Figure 18) enumerates the possible conditions, “handle_position = down ∧ NC(handle_position)” and “handle_position = up ∧ NC(handle_position)”, where “NC(handle_position)” reflects the assumption that the position of the handle does not change [89]. Requirements “$R_{21}$” and “$R_{22}$” talk about an immediate, rather than temporal, response of the system to the stimulus coming from the handle. The resulting vector relation table (Figure 19) thus does not contain any additional variables. An important property of a table is to be proper - that is, to characterize mutually disjoint situations. The LGS knows two situations: when the pilots’ handle in the cockpit stays (1) down and (2) up. The LGS environment guarantees mutual disjointness of these situations.

Assessing the tabular relations approach with respect to the criteria of section 2:

- **Audience**: the tabular notation is formal and requires mathematical qualification both from the specifiers and the readers.
- **Abstraction**: the tabular notation operates at the level of program variables. Tables specify programs through specifying how the programs’ executions affect the variables’ values under mutually disjoint preconditions.
- **Method**: the approach does not assume a particular requirements engineering method.
- **Tool**: we found no tools supporting tabular relations.
- **Traceability**: the approach works on top of program variables, so there is conceptual traceability between tables and the programs they specify.
- **Coverage**: the tabular notation focuses on specification of functional requirements. The approach is equipped with 10 classes of tables for this purpose [88].
- **Scope**: the approach can cover all aspects of requirements, including both system and environment aspects.
- **Verification**: verification techniques that work with tabular relations include automated consistency checking [45] and test oracles generation [91].
- **Semantics**: the semantics of tabular relations is formally defined [88], [89].
5.5 Programming-language-based

Discussion of programming language-based approaches consists of two recent advances in this field: multirequirements method [67] and its successor, seamless requirements [77].

5.5.1 Multirequirements. The multirequirements method treats requirements as compilable pieces of the future program-to-be. The pieces, being merged by special tools (to be developed), form an initial program blueprint. Given that, the method has limited consideration of environment-controlled properties: resulting programs cannot assume anything about phenomena they cannot control. The method, however, makes it possible to constrain software-controlled environment-visible phenomena: Design by Contract, on which the method relies, provides necessary specification mechanisms.

The first principle of the method encourages development of individual requirements incrementally on several layers, including the following three: formal, graphical, natural-language. This principle targets different classes of project stakeholders — formal descriptions for advanced programmers and for the quality-assurance team, natural-language ones for non-software-technical customer representatives, and diagrams for communication and decision purposes.

The following principles of the multirequirements method reflect its abstraction level:

(3.3) Model systems through object-oriented techniques: classes as the basic unit of decomposition, inheritance to capture abstraction variants, contracts to capture semantics.

(3.4) Use an object-oriented language (in the present discussion, Eiffel) to write the formal layer according to the principles of 3.3.

(3.5) Use the contract sublanguage of the programming language as the notation for the formal layer.

(3.6) As the goal is to describe models, not implementations, ignore the imperative parts of the programming language (such as assignment).

The multirequirements method relies on Design by Contract. The method captures requirements’ behavioral aspect through extensive use of contracts which then become a part of the resulting program. This is particularly straightforward when contracts are first-level artifacts of the language itself.

The method defines the important concepts of up- and down-traceability:

- **Up-traceability** is the property that every element of every artifact of a project follows from some element of the requirements.
- **Down-traceability** is the property that for every requirement at least one artifact follows from it.

One of its principles, however, uncovers also the notion of traceability between the layers of discourse:

(3.9) Enforce and assess traceability between the layers and all products of the requirements process, and between requirements and other product artifacts, both down and up.

Eiffel Information System (EIS) tool partially enforces the traceability between the layers. The multirequirements method does not provide any specific guidance on how exactly to enforce up- and down-traceability. The method does not support non-functional requirements, since semantically it only maps to Design by Contract plus the programming language used in the formal layer. The EIS tool supports consistency between representation layers, and AutoProof tool [115] supports static verification of resulting programs against contracts that come from requirements. The method, however, does not provide any guidance nor tools for generating program blueprints from requirements. The multirequirements method does not support verification of requirements one-by-one. Design by Contract methodology assumes verification of implementation features and classes against all the
associated contracts at once. The method also rules out verification of legacy programs not equipped with contracts.

As an illustration, the application of the multirequirements method to requirements “$R_{21}$” and “$R_{22}$” results in the following formal representation in Eiffel:

```eiffel
class LGS
invariant
  r21: (handle_position = down) implies (gear_status ≠ retracting)
  r22: (handle_position = up) implies (gear_status ≠ extending)
end
```

where expressions $r21$ and $r22$ represent assertions’ tags. These tags may be helpful for debugging the corresponding assertions’ violations. Application of the multirequirements method to requirements “$R_{11bis}$” and “$R_{12bis}$” results in the following formal representation in Eiffel:

```eiffel
class LGS
feature
  main
    do
      ensure
        r11_bis: (old handle_position = up and handle_position = up) implies (gear_status = retract)
        r11_bis: (old handle_position = up and handle_position = up) implies (door_status = closed)
        r12_bis: (old handle_position = down and handle_position = down) implies (gear_status = extended)
        r12_bis: (old handle_position = down and handle_position = down) implies (door_status = closed)
      end
end
```

Work [79] contains a detailed application of the multirequirements method to a well-known example of a cyber-physical system.

5.5.2 Seamless requirements. The seamless requirements method inherits the multilayer philosophy of multirequirements and supports the idea of using programming language at the formal layer. Seamless requirements modify multirequirements in the following aspects:

- An individual requirement serves as an abstract data type (ADT) axiom rather than a building block for the future program.
- Routines with pre- and post-conditions [76] constitute the formal layer.
- Comments on the routines constitute the natural-language layer.

As an immediate consequence, requirements documents take the form of object-oriented classes containing the requirements routines. These modifications affect the following aspects of the requirements process:

Environment properties It is possible to axiomatize and reuse arbitrary environment concepts in requirements classes.

Level of abstraction The method relies on operational abstraction and rejects principles (3.5) and (3.6) of the multirequirements method.

Associated method Support for primitive assert statements makes any programming language suitable for applying the seamless requirements method.
Traceability tool support  The method takes advantage of IDEs’ features such as “Show clients” or “Show suppliers” for traceability management, since requirements coexist with program sources in a single namespace.

Verifiability  The method enables verification of requirements one by one, since a requirement is a routine. In particular, seamless requirements are verifiable with AutoProof [115], the prover of Eiffel programs. It also supports verification of legacy programs, since requirements exist separately from implementations.

Other aspects of the multirequirements method pervade seamless requirements as well. The reliance on operational abstraction enables use of looping constructs for specification of embedded systems’ temporal properties [75]. In verification through contract-based static verification seamless requirements serve as specification drivers [76] that assist development of complete contracts. In the world of testing they serve as parameterized unit tests (PUT’s) [114] and enable formal specification of programs in the absence of contracts.

As an illustration, the following formal representation of the “R22” requirement is provided in the following:

```
R22
require handle_position = up
do main
ensure gear_status ≠ extending
end
```

This representation is submittable to a Hoare-logic-based prover AutoProof [115]. The prover accepts the “R22” routine only if the “main” feature of the “LGS” class has a strong enough contract.

The seamless requirements method also allows the specification of temporal properties:

```
R12_bis
require handle_position = up
local steps: INTEGER
do
  from steps := 0
  until (gear_status = extended and then door_status = closed) or else
    steps = steps.max_value
  loop
    main
    steps := steps + 1
  end
ensure gear_status = extended
door_status = closed
end
```

The steps local variable bounds verification of the “R12_bis” routine through the steps = steps.max_value exit condition. Because the “main” routine has no way to change the value of the read-only handle_position variable, one may safely assume that it will have the same value in the post-state than in the pre-state.

Assessing both approaches with respect to the criteria of section 2 leads to the following:

- **Audience:** the three-layers representation makes them readable to different classes of project stakeholders. It is not always clear, however, how to graphically represent such requirements (e.g., the ones from the LGS example).
• **Abstraction**: the approaches are best suited at the level of program designs and contracts.
• **Method**: the approaches do not assume a particular requirements engineering method.
• **Tool**: the EIS (Eiffel Information System) tool partially supports the process. It provides a mechanism for linking development objects of Eiffel projects with external information resources, such as Web pages, PDF documents, and Microsoft Word documents.
• **Traceability**: the approaches enforce pairwise traceability between requirements expressed in different notations by making the very notion of a requirement multinotational. Traceability between requirements and implementations still needs to be established by additional tools.
• **Coverage**: the approaches focus on functional requirements.
• **Scope**: the approaches focus on system aspects without any regard to environment aspects.
• **Verification**: the approaches rely on contracts for the formal layer, which allows for verification of candidate implementations against these contracts. It is not clear, however, how to use the approach for verification of non-modifiable legacy software.
• **Semantics**: the formal semantics of multirequirements corresponds to that of contracts as defined in [66].
• **Seamlessness**: the approach makes it possible to execute the entire software development lifecycle using the same set of notations and tools – the implementation programming language and the underlying integrated development environment (IDE), respectively.

6 RESULTS AND DISCUSSION

To draw conclusions from the present study, we first list limitations (6.1), then present a summarized table of results (6.2), and finally we explore some of the questions that have been addressed:

• Should the elicitation process start with an informal or semi-formal notation (6.3)?
• Is a seamless approach better or worse than a mix of formal and semi-formal notations (6.4)?
• What are the respective merits of natural language and graphical notations for requirements (6.5)?
• What is the current state of tools support for requirements engineering (6.6)?
• What is the current state of education in formal approaches to requirements (6.7)?

6.1 Limitations

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

• The choice, whenever possible, of a running example (the Landing Gear System) to illustrate the concepts. An alternative would have been to include a multitude of small examples to highlight individual points. The case for a single example is clear: it makes it easier to compare approaches.
• The nature of that example, a reactive system. An alternative would have been an enterprise-style system (accounting, 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 various methods.
• The LGS example, however, does not include concurrency, which prevents approaches such as Petri Nets or FSP/LTSA from showcasing some of their key properties.

6.2 A summary of the results

Table 4 presents the key conclusions in tabular form, ordered by category (from section 5), then alphabetically within each category. For non-binary criteria, the table uses these notations:

• System vs Environment:
– S: the approach can be only used to model the system
– B: the approach can be used to model both system and environment

• Prerequisites:
  – 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:
  – L: low level of abstraction
  – H: high level of abstraction
  – B: Both low and high level of abstraction

6.3 Informal versus Semi-formal versus Formal

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. [51] concludes that techniques providing a high degree of guidance and process description are critical to achieve successful results. [116] concludes that higher-level abstractions for requirements specification and analysis are critical success factors.

Formal methods have produced a number of industrial success stories, but their spread remains modest as assessed against the vast majority of projects using classical natural-language-based techniques. This survey may provide insight on how to extend that spread.

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 and, when necessary, for a binding legal contract as to what the system is supposed to do.

This counter-argument (in favor of formal methods) also has its limits. The rigor and precision of formal methods is not an excuse for ignoring the need to understand what stakeholders want. After all, 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 [60], 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 good 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 [122] and [56]) 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, possessing both expertise in the method and an ability to relate to ordinary project stakeholders. Proponents of
Formality in software requirements

formal methods often complain about the reluctance of stakeholders to use mathematical reasoning. Complaining does not need 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 practice of the database community may provide guidance. In the design of databases, the initial phase (before the switch to an implementation that often uses the relational model, another notation with a solid mathematical basis[23]), typically relies on a graphical semi-formal notations such as entity-relationships diagrams [22], which have a precise semantics but are flexible and intuitive enough for initial design and can be explained to non-IT-expert stakeholders. This

Table 4. Evaluation summary
experience shows that, with a proper process in place, there is no reason to fear systematic rejection of formal or semi-formal approaches.

6.4 Seamless versus conventional

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 size, and it is difficult to maintain consistency.

An alternative approach is to use seamless development, relying on a single set of concepts and notations throughout; Eiffel in particular was designed as a language that can cover not only the programming part but also design and, through its more abstract constructs, requirements. In such an approach there is a continuum from requirements to design to code, each step adding to the previously developed model, making it more concrete and closer to an actual program. One of the principal expected benefits is full traceability between requirements, design, code and other software artifacts. The “multirequirements” approach [67] extends the concept of seamlessness by using several complementary notations such as English, a formal notation or programming language, and a graphical notation, with the corresponding descriptions being kept in sync (the most formal of the versions serves as the reference).

The idea of using a programming language for requirements often meets with the reaction that most programming languages are imperative and all are implementation-oriented and hence will jeopardize the needed emphasis of requirements on “what” rather than “how” — the Abstraction criterion in the present survey. This concern, however, is not justified. Programming languages describe more than implementation, and when applying them to requirements we may ignore their imperative aspects (although some of the authors’ work develops a pure-requirements approach that actually takes advantage of imperative features [75–79]). Some of the major contributions of modern programming languages, the result of decades of progress and particularly of the concepts of object technology, equip them with the ability to describe big things. The notions of module/package, class, inheritance, information hiding, interface, genericity are just examples of these contributions. While by definition the “things” being described are by default programs, these scaling-up techniques introduced by programming languages are applicable to the modularization of many other kinds of formal texts, including requirements.

These techniques coming from programming languages are in fact the only ones known to scale up to extremely large systems, such as a program of many millions of lines of code. Ordinary mathematical notation is not designed for that purpose: typically mathematical statements extend over one or a few lines; to describe the relations between them, and the overall structure of a theory as introduced in an article or book, one has to resort to natural language. (This observation even applies to almost purely formal mathematical texts such as Whitehead and Russell’s *Principia Mathematica.*) Notations for formal requirements, to be practical, need scaling-up capabilities; they can get them, ready for use, from programming languages with strong modular constructs. This is the vision behind Eiffel, with its full range of object-oriented modularization techniques, plus non-imperative specification techniques of Design by Contract, eminently applicable, beyond programs, to requirements of large systems.

Starting from the requirements-ready part of a programming language and retaining the same notation throughout the remaining software activities of design, implementation etc. presents the additional advantage of removing or narrowing gaps (“impedance mismatches”) between successive steps. The practical process of software development, whether “waterfall” or “agile” in principle, is inevitably back-and-forth, with design and implementation forcing revisions to requirements.
Too often issues or simply new ideas encountered during implementation lead to changes that conceptually are requirements change, but do not get reflected back into the requirements document or user stories because of the extra burden of converting back and forth between completely different frameworks and notations (typically a programming language for programs and natural language for requirements). If everything is integrated in a single notation it becomes much more realistic to keep everything in sync, with great advantages for traceability, debugging and maintenance. (This is the “single product principle” or “single model principle” of [65].)

While seamless development runs contrary to the traditional emphasis on separation and concerns, it emphasizes the fundamental unity of software concepts throughout the lifecycle. In that view requirements are first-class citizens of the software world, on a par with other other artifacts such as code, designs and tests, and susceptible to the same rules and techniques.

More work, in particular empirical, remains necessary to question and validate the seamless approach to formal and informal requirements:

- Does seamlessness help make requirements useful for stakeholders with widely different backgrounds?
- What are the concrete traceability benefits?
- How much does seamless development reduce documentation overhead?
- To what extent does it support requirements maintenance and reuse?
- Tools: in a seamless approach, are program development environments enough, or do requirements still call for specific tools?
- How the processes of specifying requirements and implementing them could reinforce each other?

### 6.5 Textual versus graphical

For the sake of quality assessment, the notation used to express requirements should be as close as possible to the domain expert’ habits. Traditional RE (Requirements Engineering) is split into two worlds: a formal world and a natural-language world. In RE, most of the works addressed to requirements elicitation are considering natural language. Indeed, by implying several stakeholders, there is a real need to communicate. Natural language is hence the most universally used notation for this purpose.

Natural language does not necessarily mean informal. In section 5.1, works that aim at formalizing natural-language requirements have been presented. These approaches have the main advantage to be easy to handle by most kind of stakeholders.

Two types of representations can be used in the aforementioned approaches: textual ones and graphical ones. The relevance of representation is mainly a matter, for the intended audience, of habits and expressiveness. Indeed, the general purpose tools must offer a representation easy to understand by a non-expert, and even experts tools should be as close as possible to their habits.

SysML for example allows users to express requirements as model artifacts, which can be easily linked to other model elements, used by different domains experts. KAOS or i* approaches make possible to graphically represent relationships between requirements. The main purpose of these graphical approaches is to emphasize relationships and traceability between requirements, that is one of the concerns for the users of these approaches. Requirements themselves in these approaches are represented as not formalized natural-language requirements. Approaches based on graph and automata can provide a more formal graphical representation. Even if they may be more accessible to non-specialists than formal methods and textual mathematical approaches for example, contrary to aforementioned graphical methods these approaches proposed to formally express requirements.
However, the most used representation for requirements is still the textual representation, both for formal or informal natural-language representations. While the representation of the relationships between requirements appears to be simpler from a graphical point of view, the requirements themselves are undoubtedly more readable in a textual formalism. It should thus be noticed that approaches addressed to all stakeholders (such as Doors, SysML, KAOS or i*) still retain a textual representation of the requirements themselves. The concerns of the target audience appear to be the main reason for choosing a type of representation to ease the acceptance, in the same way that the choice of tool support.

While the maturity of tools to support both textual and graphical representation has reached an industrial usage, some questions still remain:

- How can requirements best combine graphical and textual representations? And incidentally what is the status of existing tools in terms of synchronisation.
- Does it make sense (and is it always possible) to translate textually a requirement model (e.g., an architecture description being considered as a requirement for those who provide the components of this architecture)?

### 6.6 Tool support

The authors of [28] showed in 2011 that only a few tools, out of 94 they studied, were adapted to modeling (42%) and requirements management (39%). In 2019, the request “requirement engineering tools” on Google mainly provides links to requirements management tools. We can deduce that in the last few years efforts have been expanded on tools for requirements management, which respond to an industrial need of elicitation, change tracking, traceability, etc., but do not consider the formalization of requirements nor the benefits of their formal expression. Nowadays the most used tools offer ergonomic interfaces which are easy to handle. From general purpose ones ([108], [97], [109],...) to formal ones ([94], [115]), all the approaches mentioned in this survey are supported by tools providing graphical interfaces.

Most of the general purpose approaches are fully supported by modeling tools allowing elicitation, description or management of requirements. Industrial tools like DOORS [49] or Reqtify [108] provide functionalities to trace requirements, link different kinds of artifacts, and give to stakeholders intuitive interfaces. KAOS or i* are supported by tools that are more dedicated to the elicitation of requirements through goals, and that support graphical models ([97]). SysML and URML are integrated into tools like ([109]) that permit to collect of requirements and their refinement, making hierarchies of requirements and structuring them to provide models of requirements. In these approaches, a requirement is characterized by properties describing its informal semantics, some being under the form of notes expressed in natural language. These tools are rich, clear, intuitive and complete enough to answer stakeholders needs and allow them to get an interesting level of abstraction and to reach their objectives. But they stand at a contemplation level: they do not deal with formalization, inference or deductive reasoning.

As these approaches are more addressed to the upstream phases of the software project lifecycle, an emphasis should be put on the integration of these tools inside more complete tooling. As presented in [87], the use of a single model that groups several artifacts intrinsically should lead avoiding inconsistencies between requirements, specification and the system itself. AutoTest framework [68] already takes advantage of the Pex technology [113] to automate testing of arbitrary programs equipped with contracts, thus potentially supporting both the multirequirements and the seamless requirements methods. It should also be possible to employ the AxiomMeister technology [112], that currently works in the .NET world, for inference of potential requirements based on the source code.

---

15https://www.capterra.com/requirements-management-software/
The tools that have some formalization capabilities are very specific, meant for experts, and most of the time disconnected from tools supporting other activities as modeling and traceability. This implies that the formalization of requirements is mainly used in critical systems when proof of correctness and formal validation are absolutely necessary. So systems that may benefit from formalization (and thus from formal verification) do not. Therefore, if the need for tool support is obvious and commonly admitted, the supply of an integrated tool would be interesting or even necessary. The expression of requirements from their modeling to their formalization should be supported, especially if this latter can be automatically obtained.

These reflections on tool support raised some open research questions:

- Is there a general pattern to express textual requirements (is there a unique grammar)? Or is it depending on each domain area?
- How do we (as automatically as possible) measure the quality of a requirement?
- Should these tools provide different viewpoints adapted to their stakeholders’ concerns, allowing them to manipulate different kinds of requirements expression (NL, formal, ...)?
- To what extent can the functioning code be generated automatically from a formal representation of requirements? How desirable is this?
- Wouldn’t integrated tools be a notable support to requirements education?

6.7 Education

Interest in formal approaches to express requirements and, more generally, to design software has been progressively growing for the last few decades. Consequently, more and more educational institutions paired their research effort with a pedagogical effort in software engineering tracks [59]. This synergy, combined with the increasing interest of industry in formal approaches, grew 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, being it a big corporation or their newly funded startup. The literature on pedagogical aspects of formal methods is vast [29, 36], in particular on approaches to software engineering courses built on strong mathematical basis [37]. Some resistance has been documented while deploying such approaches, and the issue of motivation has also been investigated [86, 96].

It is far beyond the objectives of this work to be exhaustive to this regard. We will only provide here an overview of general trends in formal method education, and describe examples of pedagogical use of the formalism and approaches described in this survey.

The general purpose requirements formalisms are not often used to teach requirements engineering. If SysML is used, it is mainly in a larger scope of MBSE (Model-Based System Engineering) education ( [82]). Some other approaches like Kaos or i* are essentially used at requirements elicitation step ([74]), notably to students having no knowledge in formal methods ([26]) or to complete formal method by domain specific ones ([52]). For example, there is a workshop series dedicated to i* teaching since 2015 (iStar@CAISE2015, iStar@ER2017).

The idea of natural-language-based approaches is, inter alia, to avoid the difficulty of teaching a new approach. Indeed, by providing an interface between natural language and the formal representation of requirements, these approaches are based on the presupposed knowledge of natural-language requirements. The teaching of expressing requirements with natural language is probably the first step of any requirements engineering course. In some works [40, 129], difficulties in teaching requirements elicitation are highlighted, and solutions are proposed to overcome them. However, these difficulties are not linked to the use of natural language for requirements, but are about requirements qualities (such as completeness, consistency, etc) and the emphasis is put on how to provide good requirements – e.g., in [122], the authors propose “good practices” for requirements.
Graph and automata have always been a popular representation for students [12, 39]. Despite their mathematical foundations, they are graphical, easy to understand, and not difficult to produce. This apparent user-friendliness often leads to misunderstanding and errors because of the lacks of clear execution semantics, as reported by famous articles [25, 30, 120]. In that matters, the recent progress in executable semantics and in tools animation (see previous section 6.6) is a very important progress with regards to education.

In [20] experiences are reported in teaching formal methods, in particular JML [34] and B [4]. Design and delivery of courses aiming at developing skills of model construction and analysis by use of notations such as VDM-SL and VDM++ are presented in [57]. The motivation problem has been here improved by using examples from industrial projects, and by using an industrial-strength tool set. Concurrency theory and FSP/LTSA have also been used in teaching and documented [8]. Among all formalisms for concurrency, CSP has been the one with more widespread applications, both industrial and educational [99]. More interactive ways of introducing formal specifications have been experimented, for example in [110], where an on-line tutorial has been designed in order to help students in the transition from Z to Alloy, considered the latter more practical to use due to the existence of the Alloy Analyzer.

Programming language-based approaches to requirements have just started emerging, which is why there is no yet any existing data to refer to on this matter. It is still possible, however, to speculate on potential ways of teaching this family of approaches. Programming language-based approaches rely most of the time on DbC (Design by Contract). Both approaches mentioned in subsection 5.5 are easy to explain to someone who is familiar with DbC: in the original multirequirements method, requirements take the form of contracted excerpts from the final program, while in the seamless requirements method they take the form of contracted routines expressed in terms of the final program.

In the field of education, a number of questions remain outstanding, which we cannot answer conclusively here:

- Are extensions to existing programming courses relying on DbC a suitable approach to teach both basic programming and formal methods with an emphasis on quality?
- Should the idea of seamless development introduced into the way in which software engineering is taught?
- Why is there so little emphasis on requirements in regular Software Engineering curricula?
- How relevant is introducing formal notations in introductory courses?

### 6.8 Conclusion: what role for formal approaches to 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, examining them on a scale assessing their degree of formalism: 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 continue to cause endless 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 many stakeholders do not understand formal texts. There is truth is 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.

, Vol. 1, No. 1, Article 1. Publication date: November 2019.
As an example of the limits of classic but simplistic views, consider the “stakeholders do not 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 car models 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, international transfers etc., no single stakeholder is an expert in all these disciplines.

One suspects that often that 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 comments of that kind: 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 Ohm’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 2. 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 aspects of that system (in the same way our car marketer must understand what is new in the latest model). Here the notion of view comes into the picture. Requirements of ambitious systems may have to rely on several views, each 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 is, 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 work [67] develops this idea further, proposing to write requirements in a combination of natural-language, graphical and formal notations, the formal one expressed in Eiffel and 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-purpose techniques such as UML or SysML have enjoyed widespread use, showing that when the benefits are clear users do not hesitate to learn highly technical approaches.

The last comment more generally shows the way to progress in the formal-versus-informal debate. Ideological discussions should yield to pragmatic considerations. Techniques will gain acceptance if they produce substantial, tangible benefits commensurate with the effort they require. Two crucial conditions of that success are:

• Tools: even the most impressive method and elegant notation will not catch on without automated support. Good tools relieve programmers from mundane aspects, flag errors and inconsistencies, and scale up to large systems.

• Education: software engineering education typically introduces too many 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 6.4 discussed the arguments for a more seamless approach, which threads 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 epitomize the relationship of 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 are not a panacea, and they 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 this 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. His insightful comments helped us additionally to a better understanding of how to handle such examples in FSP/LTSA. We would also like to thank Thuy Nguyen who has carried out a formalization of the LGS in Form-L.

REFERENCES

[1] 1998. IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830-1998 (1998), 1–40. DOI: http://dx.doi.org/10.1109/IEEESTD.1998.88286
[25] Arnaud Cuccuru, Chokri Mraidha, Francois Terrier, and Sbastien Grard. 2007. Enhancing UML Extensions with Operational Semantics Behaved Profiles with Templates. In Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS’07). Springer-Verlag, Berlin, Heidelberg, 271–285. http://dl.acm.org/citation.cfm?id=2394101.2394127

[26] Fabiano Dalpiaz. 2015. Teaching Goal Modeling in Undergraduate Education. In Proceedings of the 1st International iStar Teaching Workshop co-located with the 27th International Conference on Advanced Information Systems Engineering (CAiSE 2015), 1–6.

[27] Anne Dardenne, Axel van Lamsweerde, and Stephen Ficks. 1993. Goal-directed requirements acquisition. Science of Computer Programming 20, 1 (1993), 3–50. DOI: http://dx.doi.org/10.1016/0167-6423(93)90021-G

[28] Juan M Carrillo de Gea, Joaquín Nicolás, José L Fernández Alemán, Ambrosio Toval, Christof Ebert, and Aurora Vizcaíno. 2011. Requirements Engineering Tools. IEEE Software 28, 4 (July 2011), 86–91. DOI: http://dx.doi.org/10.1109/MS.2011.81

[29] Neville Dean and Michael Gerard Hincheey. 1996. Teaching and Learning Formal Methods. Academic Press.

[30] Trung Dinh-Trong, Arnor Solberg, Sudipto Ghosh, and Robert B. France. 2006. Model-Driven Development Using UML 2.0: Promises and Pitfalls. Computer 39, 2 (2006), 59–66. DOI: http://dx.doi.org/10.1109/MC.2006.65

[31] Eugene Durr and Jan van Katwijk. 1992. A formal specification language for object-oriented designs. In CompEuro 1992 Proceedings Computer Systems and Software Engineering. 214–219. DOI : http://dx.doi.org/10.1109/CMPEUR.1992.218511

[32] Moritz Eysholdt and Heiko Behrens. 2010. Xtect: implement your language faster than the quick and dirty way. In Proceedings of the Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010. ACM, 307–309. DOI: http://dx.doi.org/10.1145/1869542.1869625

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

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

[35] J. M. Carrillo de Gea, J. Nicolás, J. L. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno. 2011. Requirements Engineering Tools. IEEE Software 28, 4 (July 2011), 86–91. DOI: http://dx.doi.org/10.1109/MS.2011.81

[36] Jeremy Gibbons and José Nuno Oliveira (Eds.). 2009. Teaching Formal Methods. TFM 2009. Lecture Notes in Computer Science, Vol. 5846. Springer, Berlin, Heidelberg.

[37] Paul Gibson and Dominique Méry. 1998. Teaching Formal Methods: Lessons to Learn. In Proceedings of the 2nd Irish Conference on Formal Methods (IW-FM ’98). BCS Learning & Development Ltd., Swindon, UK, 56–68.

[38] Rainer Gmehlich, Katrin Grau, Felix Loesch, Alexei Iliasov, Michael Jackson, and Manuel Mazzara. 2013. Towards a formalism-based toolkit for automotive applications. In 2013 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE), 36–42. DOI: http://dx.doi.org/10.1109/FormaliSE.2013.6612275

[39] Michael T. Grinder. 2002. Animating Automata: A Cross-platform Program for Teaching Finite Automata. SIGCSE Bull. 34, 1 (Feb. 2002), 63–67. DOI: http://dx.doi.org/10.1145/563517.563364

[40] Thomas Hainey, Thomas M Connolly, Mark Stansfield, and Elizabeth A Boyle. 2011. Evaluation of a game to teach requirements collection and analysis in software engineering at tertiary education level. Computers & Education 56, 1 (2011), 21–35. DOI: http://dx.doi.org/10.1016/j.compedu.2010.09.008

[41] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 3 (June 1987), 231–274. DOI: http://dx.doi.org/10.1016/0167-6423(87)90035-9

[42] Jameleddine Hassine. 2010. AsmL-Based Concurrency Semantic Variations for Timed Use Case Maps. In International Conference on Abstract State Machines, Alloy, B and Z, Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves (Eds.). Springer, 34–46.

[43] Jameleddine Hassine and Daniel Amyot. 2016. A Questionnaire-based Survey Methodology for Systematically Validating Goal-oriented Models. Requir. Eng. 21, 2 (jun 2016), 285–308. DOI: http://dx.doi.org/10.1007/s00766-015-0221-7

[44] Jameleddine Hassine and Daniel Amyot. 2017. An Empirical Approach Toward the Resolution of Conflicts in Goal-oriented Models. Softw. Syst. Model. 16, 1 (Feb. 2017), 279–306. DOI: http://dx.doi.org/10.1007/s10270-015-0460-6

[45] Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. 1996. Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology (TOSEM) 5, 3 (1996), 231–261. DOI: http://dx.doi.org/10.1145/234426.234431

[46] Jonas Helming, Maximilian Koegel, Florian Schneider, Michael Haeger, Christine Kaminisk, Bernd Bruegge, and Brian Berenbach. 2010. Towards a unified Requirements Modeling Language. In 2010 Fifth International Workshop on Requirements Engineering Visualization. 53–57. DOI: http://dx.doi.org/10.1109/REV.2010.5625659

[47] Charles Antony Richard Hoare. 1978. Communicating Sequential Processes. Commun. ACM 21, 8 (Aug. 1978), 666–677.
[48] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. 2002. An Authoring Tool for Informal and Formal Requirements Specifications. In Fundamental Approaches to Software Engineering, Ralf-Detlef Kutsche and Herbert Weber (Eds.). Number 2306 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 233–248. DOI: http://dx.doi.org/10.1007/3-540-45923-5_16

[49] IBM. 2015. Rational Doors. (2015). http://www-03.ibm.com/software/products/ratidoor

[50] IBM. 2016. Rational Rhapsody 8.1.5. (may 2016). http://www-03.ibm.com/software/products/en/ratifadesiforsystengti

[51] Wong Cheng In and Marie Hélène LinLee. 1994. Informal, semi-formal, and formal approaches to the specification of software requirements. Ph.D. Dissertation. https://open.library.ubc.ca/cIRcle/collections/831/items/1.0051578

[52] Fuyuki Ishikawa, Kenji Taguchi, Nobukazu Yoshioka, and Shinichi Honiden. 2009. What Top-Level Software Engineers Tackle after Learning Formal Methods: Experiences from the Top SE Project. In Teaching Formal Methods: Second International Conference, TFM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings, Jeremy Gibbons and José Nuno Oliveira (Eds.). Springer Berlin Heidelberg, 57–71. DOI: http://dx.doi.org/10.1007/978-3-642-04912-5_5

[53] Daniel Jackson. 2006. Software Abstractions: Logic, Language, and Analysis. The MIT Press.

[54] Michael Jackson. 2001. Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[55] Bertrand Jeannet and Fabien Gaucher. 2015. Debugging real-time systems requirements: simulate the “what” before the “how”. In Embedded World Conference, Nurnberg, Germany. https://pdfs.semanticscholar.org/ef5e/f67853d58ca06090395f92ed31d29d4d6.pdf

[56] Phillip Laplante. 2017. Requirements Engineering for Software and Systems. Auerbach Publications.

[57] Peter Gorm Larsen, John S. Fitzgerald, and Steve Riddle. 2009. Practice-oriented courses in formal methods using VDM. Formal Aspects of Computing 21, 3 (May 2009), 245–257. DOI: http://dx.doi.org/10.1007/s00165-008-0068-5

[58] Feng-Lin Li, Jennifer Horkoff, Alexander Borgida, Giancarlo Guizzardi, Lin Liu, and John Mylopoulos. 2015. From Formality in software requirements 1:45 Software Tools for Technology Transfer. Springer, 108–125.

[59] Shaoying Liu, Kazuhiro Takahashi, Toshinori Hayashi, and Toshihiro Nakayama. 2009. Teaching Formal Methods in the Context of Software Engineering. SIGCSE Bull. 41, 2 (June 2009), 17–23. DOI: http://dx.doi.org/10.1145/1595453.1595457

[60] Robyn R. Lutz. 1993. Analyzing software requirements errors in safety-critical, embedded systems. In [1993] Proceedings of the IEEE International Symposium on Requirements Engineering. 126–133. DOI: http://dx.doi.org/10.1109/ISRE.1993.324825

[61] Jeff Magee and Jeff Kramer. 1999. Concurrency: State Models & Java Programs. John Wiley & Sons, Inc., New York, NY, USA.

[62] Amel Mammar and Régine Laleau. 2017. Modeling a landing gear system in Event-B. International Journal on Software Tools for Technology Transfer 19, 2 (01 Apr 2017), 167–186.

[63] Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau. 2010. An Event-B formalization of KAOS goal refinement patterns. [Research Report] TR-LACL-2010-1, LACL.

[64] Bertrand Meyer. 1985. On Formalism in Specifications. Software, IEEE 2 (02 1985), 6 – 26. DOI: http://dx.doi.org/10.1109/MS.1985.229776

[65] Bertrand Meyer. 1988. Object-Oriented Software Construction. Vol. 2. Prentice-Hall, Inc.

[66] Bertrand Meyer. 2003. A framework for proving contract-equipped classes. In International Workshop on Abstract State Machines. Springer, 108–125.

[67] Bertrand Meyer. 2013. Multirequirements. Modelling and Quality in Requirements Engineering (Martin Glinz Festschrifft) (2013).

[68] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. 2009. Programs that test themselves. Computer 42, 9 (2009), 46–55. DOI: http://dx.doi.org/10.1109/MC.2009.296

[69] Luisa Mich. 1996. NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. Natural Language Engineering 2, 2 (jun 1996), 161–187. DOI: http://dx.doi.org/10.1017/S1351324996001337

[70] Robin Milner. 1982. A Calculus of Communicating Systems. Springer-Verlag, Berlin, Heidelberg.

[71] Marvin L. Minsky. 1967. Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

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

[73] Seong-ick Moon, K. H. Lee, and Doheon Lee. 2004. Fuzzy branching temporal logic. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 34, 2 (April 2004), 1045–1055. DOI: http://dx.doi.org/10.1109/TSMCB.2003.819485
Formality in software requirements

[98] Alexander Romanovskiy and Martyn Thomas. 2013. Industrial Deployment of System Engineering Methods. Springer Publishing Company, Incorporated.

[99] Andrew William Roscoe. 1997. The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[100] Jean-François Roy, Jason Keailey, and Daniel Amyot. 2006. Towards Integrated Tool Support for the User Requirements Notation. In System Analysis and Modeling: Language Profiles: 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 - June 2, 2006, Revised Selected Papers, Reinhard Gotzheim and Rick Reed (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 198–215. https://doi.org/10.1007/11951148_13

[101] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education.

[102] Richard E. Schneider and Dennis M. Buede. 2000. Properties of a High Quality Informal Requirements Document. INCOSE International Symposium 10, 1 (July 2000), 352–359. DOI:http://dx.doi.org/10.1002/j.2334-5837.2000.tb00397.x

[103] Steve Schneider, Helen Treharne, and Heike Wehrheim. 2014. The behavioural semantics of Event-B refinement. Formal Aspects of Computing 26, 2 (01 Mar 2014), 251–280. DOI:http://dx.doi.org/10.1007/s00165-012-0265-0

[104] William Scott and Stephen Cook. 2004. A Context-free Requirements Grammar to Facilitate Automatic Assessment. In Ninth Australian Workshop on Requirements Engineering (AWRE 2004): Proceedings. https://www.researchgate.net/profile/Stephen_Cook10/publication/228853367_A_context-free_requirements_grammar_to_facilitate_automatic_assessment/links/0e96052ed5f594e0108000000.pdf

[105] John Slankas and Laurie Williams. 2013. Automated extraction of non-functional requirements in available documentation. In 2013 1st International Workshop on Natural Language Analysis in Software Engineering (NaturaLiSE). 9–16. DOI:http://dx.doi.org/10.1109/NaturaLiSE.2013.6611715

[106] Michel dos Santos Soares, Jos Vrancken, and Alexander Verbraeck. 2011. User Requirements Modeling and Analysis of Software-intensive Systems. J. Syst. Softw. 84, 2 (Feb 2011), 328–339. DOI:http://dx.doi.org/10.1016/j.jss.2010.10.020

[107] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. SIGCOMM Comput. Commun. Rev. 31, 4 (Aug. 2001), 149–160. DOI:http://dx.doi.org/10.1145/383059.383071

[108] Dassault Systems. 2016. CATIA Reqtify. (2016). https://www.3ds.com/products-services/catia/products/reqtify

[109] Sparx Systems. 2017. Enterprise Architect. (2017). https://www.sparxsystems.eu/enterprisearchitect/system-requirements/

[110] Sureyya Tarkan and Vibha Sazawal. 2009. Chief Chefs of Z to Alloy: Using a Kitchen Example to Teach Alloy with Z. Springer Berlin Heidelberg, 72–91. DOI:http://dx.doi.org/10.1007/978-3-642-04912-5_6

[111] Thomas Tilley, Richard Cole, Peter Becker, and Peter Eklund. 2005. A survey of formal concept analysis support for software engineering activities. In Formal concept analysis. Lecture Notes in Computer Science, vol 3626, Wille R. Ganter B., Stumme G. (Ed.). Springer, Berlin, Heidelberg, 250–271. DOI:http://dx.doi.org/10.1007/11528784_13

[112] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. 2006. Discovering Likely Method Specifications. In Formal Methods and Software Engineering. ICFEM 2006. Lecture Notes in Computer Science, vol 4260, Jifeng Liu and Jifeng He (Eds.). Springer Berlin Heidelberg, 717–736. DOI:http://dx.doi.org/10.1007/11901433_39

[113] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex-white box test generation for .NET. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 4966. 134–153. DOI:http://dx.doi.org/10.1007/978-3-540-79124-9_10

[114] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. ACM SIGSOFT Software Engineering Notes 30, 5 (2005), 253. DOI:http://dx.doi.org/10.1145/1095430.1081749

[115] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. 2015. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In Tools and Algorithms for the Construction and Analysis of Systems, Christel Baier and Cesare Tinelli (Eds.). Lecture Notes in Computer Science, Vol. 9035. Springer Berlin Heidelberg, 566–580. DOI:http://dx.doi.org/10.1007/978-3-662-46681-0_53

[116] Axel van Lamsweerde. 2000. Formal Specification: A Roadmap. In Proceedings of the Conference on the Future of Software Engineering. ACM, 147–159.

[117] Axel van Lamsweerde. 2001. Goal-oriented requirements engineering: a guided tour. In Proceedings Fifth IEEE International Symposium on Requirements Engineering. 249–262. DOI:http://dx.doi.org/10.1109/ISRE.2001.948567

[118] Axel van Lamsweerde. 2004. Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice. In 12th IEEE Joint International Requirements Engineering Conference, Kyoto, 4–8. DOI:http://dx.doi.org/10.1109/ICRE.2004.1335648

[119] Emilia Villani, Fabrício Junqueira, Paulo Eigi Miyagi, and Robert Valette. 2004. Petri net and OO for the modular analysis of an aircraft landing system. ABCM Symposium Series in Mechatronics 1 (11 2004), 570–579. http://www.abcm.org.br/symposium-series/SSM_Vol1/Section_IV_Discrete_Event_Dynamic_Systems/SSM_IV_08.pdf
A LGS IN FSP/LTSA

// Reaction of the LGS to the different states of the handle.
LGS_BEHAVIOR = (do_nothing -> LGS_BEHAVIOR | up -> OPEN_FOR_RETRACTION | down -> OPEN_FOR_EXTENSION),
OPEN_FOR_RETRACTION = (open -> START_RETRACTION),
START_RETRACTION = (start_retraction -> END_RETRACTION),
END_RETRACTION = (end_retraction -> START_CLOSING_RETRACTED | down -> START_CLOSING_RETRACTED),
START_CLOSING_RETRACTED = (start_closing -> END_CLOSING_RETRACTED),
END_CLOSING_RETRACTED = (end_closing -> LGS_BEHAVIOR | down -> OPEN_FOR_EXTENSION),
OPEN_FOR_EXTENSION = (open -> START_EXTENSION),
START_EXTENSION = (start_extension -> END_EXTENSION),
END_EXTENSION = (end_extension -> START_CLOSING_EXTENDED | up -> START_CLOSING_EXTENDED),
START_CLOSING_EXTENDED = (start_closing -> END_CLOSING_EXTENDED),
END_CLOSING_EXTENDED = (end_closing -> LGS_BEHAVIOR | up -> OPEN_FOR_RETRACTION),

// Fluents that model possible states of the LGS:
fluent HANDLE_IS_DOWN = <{down}, {up}>
fluent HANDLE_IS_UP = <{up}, {down}>
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}, open}>
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}>

// Specifying the LGS control handle.
CONTROL_HANDLE = (down -> INITIALLY_DOWN | up -> INITIALLY_UP),
INITIALLY_DOWN = (up -> down -> INITIALLY_DOWN),
INITIALLY_UP = (down -> up -> INITIALLY_UP).

// Modelling different modes of operation.
||LGS = (LGS_BEHAVIOR || CONTROL_HANDLE) >>
{up}. // Uncomment to model LGS with the handle pushed down.
// {down}. // Uncomment to model LGS with the handle pulled up.
assert EVENTUALLY_ALWAYS_DOWN = <> [] HANDLE_IS_DOWN
assert R11bis = [] ([] HANDLE_IS_DOWN -> <> []) (GEAR_IS_EXTENDED && DOOR_IS_CLOSED)
assert R21 = [] ([] HANDLE_IS_DOWN -> []) ! GEAR_IS_RETRACTING
assert EVENTUALLY_ALWAYS_UP = <> [] HANDLE_IS_UP
assert R12bis = [] ([] HANDLE_IS_UP -> <> []) (GEAR_IS_RETRACTED && DOOR_IS_CLOSED)
assert R22 = [] ([] HANDLE_IS_UP -> []) ! GEAR_IS_EXTENDING

B LGS IN ALLOY

abstract sig Handle {}
	sig Down, Up extends Handle {}{Down + Up = Handle}
abstract sig Door {}
	sig Closed, Opening, Open, Closing extends Door {}{Closed + Opening + Open + Closing = Door}

abstract sig Gear {}
	sig Extended, Retracting, Retracted, Extending extends Gear {}{Extended + Retracting + Retracted + Extending = Gear}

sig LGS {
    handle: Handle,
    door: Door,
    gear: Gear
}

    (gear in Retracting V gear in Extending) implies (door in Open)
    (handle in Up) implies (gear not in Extending)

pred closeDoor [lgs, lgs': LGS] {
    (lgs.door in Open) implies (lgs'.door in Closing)
    (lgs.door in Closing) implies (lgs'.door in Closed)
    (lgs.door in Opening) implies (lgs'.door in Closing)
    (lgs.door in Closed) implies (lgs'.door in Closed)
    (lgs.gear = lgs'.gear)
}

pred openDoor [lgs, lgs': LGS] {
    (lgs.door in Open) implies (lgs'.door in Open)
    (lgs.door in Closing) implies (lgs'.door in Opening)
    (lgs.door in Opening) implies (lgs'.door in Open)
    (lgs.door in Closed) implies (lgs'.door in Opening)
    (lgs.gear = lgs'.gear)
}

pred retractGear [lgs, lgs': LGS] {
    (lgs.gear in Extended) implies (lgs'.gear in Retracting)
    (lgs.gear in Retracting) implies (lgs'.gear in Retracted)
    (lgs.gear in Extending) implies (lgs'.gear in Retracting)
    (lgs.gear in Retracted) implies (lgs'.gear in Retracted)
    (lgs.door = lgs'.door)
}

pred extendGear [lgs, lgs': LGS] {
    (lgs.gear in Extended) implies (lgs'.gear in Extended)
}
(lgs.gear in Retracting) implies (lgs'.gear in Extending)
(lgs.gear in Extending) implies (lgs'.gear in Extended)
(lgs.gear in Retracted) implies (lgs'.gear in Extending)
(lgs.door = lgs'.door)
}

pred retractionSequence [lgs, lgs' : LGS] {
( lgs.gear not in Retracted) implies openDoor [ lgs, lgs' ]
(( lgs.gear not in Retracted) ∧ ( lgs.door in Open)) implies retractGear [ lgs, lgs' ]
(lgs.gear in Retracted) implies closeDoor [ lgs, lgs' ]
}

pred outgoingSequence [ lgs, lgs' : LGS ] {
(lgs.gear not in Extended) implies openDoor [ lgs, lgs' ]
(( lgs.gear not in Extended) ∧ ( lgs.door in Open)) implies extendGear [ lgs, lgs' ]
(lgs.gear in Extended) implies closeDoor [ lgs, lgs' ]
}

pred Main [ lgs, lgs' : LGS ] {
(lgs.handle in Up) implies retractionSequence [ lgs, lgs' ]
(lgs.handle in Down) implies outgoingSequence [ lgs, lgs' ]
}

run Main for exactly 2 LGS, 2 Handle, 4 Door, 4 Gear
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

R22: check {
 all lgs, lgs': LGS | ((lgs.handle in Up) and (lgs'.handle in Up) and Main [ lgs, lgs' ])
 ←→ implies (lgs'.gear not in Extending)
 } for 5

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

R12bis: check {
 all lgs1, lgs2, lgs3, lgs4 : LGS | 
((lgs1.handle in Up and lgs2.handle in Up and lgs3.handle in Up and lgs4.handle in Up) and
 (Main [ lgs1, lgs2 ] and Main [ lgs2, lgs3 ] and Main [ lgs3, lgs4 ]))
 implies (lgs4.gear in Retracted and lgs4.door in Closed))
 } for 5