Array 7 (2020) 100030

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/journals/array/2590-0056/open-access-journal

A survey on the formalisation of system requirements and their validation

Konstantinos Mokos, Panagiotis Katsaros [*]

School of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece

ARTICLE INFO

Keywords:
Requirement specification
Requirement formalisation
Semantic analysis
Model-based design
Component-based design
Formal verification

ABSTRACT

System requirements define conditions and capabilities to be met by a system under design. They are a partial definition in natural language, with inevitable ambiguities. Formalisation concerns with the transformation of requirements into a specification with unique interpretation, for resolving ambiguities, underspecified references and for assessing whether requirements are consistent, correct (i.e. valid for an acceptable solution) and attainable. Formalisation and validation of system requirements provides early evidence of adequate specification, for reducing the validation tests and high-cost corrective measures in the later system development phases. This article has the following contributions. First, we characterise the specification problem based on an ontology for some domain. Thus, requirements represent a particular system among many possible ones, and their specification takes the form of mapping their concepts to a semantic model of the system. Second, we analyse the state-of-the-art of pattern-based specification languages, which are used to avoid ambiguity. We then discuss the semantic analyses (missing requirements, inconsistencies etc.) supported in such a framework. Third, we survey related research on the derivation of formal properties from requirements, i.e. verifiable specifications that constrain the system's structure and behaviour. Possible flaws in requirements may render the derived properties unsatisfiable or not realizable. Finally, this article discusses the important challenges for the current requirements analysis tools, towards being adopted in industrial-scale projects.

1. Introduction

Requirements engineering is of vital importance in systems engineering. It consists of the elicitation, specification and management of requirements, with the aim to deliver a system that meets the stakeholders' needs.

System requirements specify how the system will meet the higher-level stakeholder requirements (also called early requirements). These requirements are a partial specification of a system solution to a design problem that is not restricted to a specific design. Since requirements are usually written using a controlled natural language (i.e. restricted in syntax and/or lexical terms), they are inevitably ambiguous. This undermines our ability to guarantee essential qualities, such as the absence of underspecified references to system entities and inconsistencies, as well as that the specification is valid for an acceptable solution (correct) and attainable. Possible flaws in system requirements imply iterated costly cycles of validation testing and corrections to the system's design, during the later phases of system development.

Therefore, the potential to properly *validate* and *refine* the system requirements, early in the development cycle, and to apply this to industrial-scale projects is a challenge that has not yet been adequately addressed in the current state of practice. Validation is associated with the problem of transforming requirements into a formal specification amenable to verification (*requirements formalisation*). This implies that all detected ambiguities and underspecified references have been resolved, while a mapping to a precisely defined semantic model of the system has been specified. We therefore assume that specification takes place with reference to a *domain model*, i.e. an ontology with precisely defined logical relationships and facts about the concepts mentioned in the requirements of a system's domain. Potential ambiguities can be eliminated if using a language of *requirement patterns* with well-defined semantics. Then, various semantic analyses can be supported, like detecting cases of missing requirements, checking the absence of contradictory specifications, discovering terms to be replaced with more concrete subclasses of a concept and so on. However, even after having applied these analyses, there is not yet any guarantee for the absence of specification flaws; though they improve the system requirements at the semantic level, they cannot render them *verifiable*. This may be happen only if the system requirements are satisfiable and realizable, something that can be decided after they have been transformed into a formal

* Corresponding author.
E-mail addresses: kmokos@csd.auth.gr (K. Mokos), katsaros@csd.auth.gr (P. Katsaros).

https://doi.org/10.1016/j.array.2020.100030
Received 1 January 2020; Received in revised form 7 May 2020; Accepted 9 May 2020
Available online 28 May 2020

specification.

This takes place, when capturing system requirements by *properties* in a logic language, for the expected behaviours and structure of a correct system design. While requirements are supposed to be independent from a particular system design, properties are expressed in terms of a *formal model* (i.e. an abstracted representation in a specification language with formal semantics) of the system under design and in fact constrain the system's design. Properties derivation can be based on a language of *property patterns*, where each pattern has been assigned semantics in a logic language. Every single requirement is covered by properties that can be derived through property patterns associated with the specific patterns of the requirement's specification and by associating the requirement's concepts to events of the system's formal model. This ensures that all requirements have a consistent interpretation with respect to the system's formal model. If the properties are not satisfied by the model, then a modified design (and model) has to be pursued or certain requirements that are not satisfied have to be refined.

This article covers all mentioned challenges of formalisation and validation of system requirements, reviews the latest state-of-the-art and discusses the existing pattern languages, the available tools and their limitations with respect to the potential of applying them in industrial-scale projects.

The rest of the article is structured as follows. The next section discusses the problems of ambiguity and underspecification in natural language requirements. Section 3 presents the recent advances on the ontology-based modeling of system requirements, their specification by means of appropriate pattern languages and the supported semantic analyses. Section 4 discusses the problem of requirement formalisation through the derivation of formal properties. This problem is related to the problem of system design and for this reason we also present the two main paradigms of model-based and component-based system design. Both design paradigms are relevant to the description of the four related tools in Section 5. Finally, Section 6 explains the limitations and the important challenges for applying a requirements formalisation/validation approach to industrial-scale projects. The last section concludes the paper and discusses some more future research prospects.

2. Ambiguity and underspecification in natural language requirements

Ambiguity in a natural language requirement stems from the possibility of interpreting it in different ways. The problem may be attributed to the syntax (e.g. ambiguous grouping of connectives) or its semantics, i.e. when there are multiple readings for a word or a vague adjunct [1], but it may be also due to neglecting some contextual information of a word. Ambiguous requirements with many interpretations allow interfacing code, usually written by different engineers, to operate under diverse assumptions [2]. The problem is exacerbated if the author is not available for clarifications, when the requirements are interpreted, during the system's implementation. This is a particularly hard problem, since there are various kinds of meaning [3] encoded in the lexical or grammatical structure of sentences. For system requirements, we focus on a sentence-meaning that is exhausted by the propositional content of sentences, which is truth-conditionally explicable (i.e. the truth of the proposition depends on something outside the language that is used [4]).

Also, a requirement may be impossible to be interpreted, when some required details are missing. In [5], this problem is referred as *incompleteness* of individual requirements and it is associated with likely incomplete and/or incorrect design of the system architecture, due to adopting potentially incorrect guesses or assumptions. As expected, incomplete requirements are not verifiable and they can imply serious safety issues in systems design. For example [6], when the verb "send" is used, two arguments are expected, the "sender" and the "recipient". If the "recipient" is omitted, it is not possible to know whether there are one or more recipients. Thus, for every predicate (verb, noun, adjective, preposition), all obligatory arguments have to be specified. A promising approach to address the incompleteness of individual requirements [5] is to adopt adequate notions of completeness, for every single type of requirement (functional, inreface, extra-fuctional etc.).

Underspecifications refer to using words that denote an entire class of objects, without a modifier specifying a concrete instance of the class [7,8]. Every such word is compatible with multiple possible semantic interpretations, thus leading to ambiguity, whose implications have been already discussed.

Industrial-scale projects are mostly based on guidelines for the way of expressing system requirements, i.e. they introduce a style in language use, for clarity. In a space system project, we encountered the following guideline:

"A requirement will contain the word shall. Equivalent expressions are allowed in exceptional cases: is required to, has to etc".

Certain words that violate verifiability (e.g. "is not allowed", "is required to be not", "must not") may be excluded.

However, these guidelines do not help to eliminate the mentioned sources of vagueness, which are inherent in all natural language specifications.

3. Semantic modeling of requirements and pattern-based specification

Natural language requirements can be formalised only through a partial loss of expressiveness. To eliminate the ambiguity of free-form syntax, a natural-like artificial language is usually used [9–14], which allows combining specification patterns (*boilerplates*) through a strictly defined syntax (context-free grammar) based on connectives with precise semantics. Boilerplates consist of *attributes* and *fixed syntax elements* such as:

$\langle system \rangle$ shall $<function>$

where "shall" is a fixed element, while $\langle system \rangle$ and $<function>$ are attributes of placeholders for user input. Boilerplates usually consist of at most three clauses: (i) the *prefix*, for specifying a stimulation or a condition, (ii) the *main* and (iii) the *suffix*, for specifying various constraints. Connectives are used in the boilerplate syntax to introduce additional entities in a specification, or for determining time, order/sequence, consequence, comparison and various types of conjunctions (coordinating, correlative, subordinating).

To avoid vague semantics, due to many word readings, *boilerplates are instantiated to specifications* by mapping attributes to class instances from an ontology with all concepts (and relationships) needed for writing requirements of a system [11,12,15–17]. Multiple ontologies are used (a requirement can refer to diverse *system domains*) that must comply with a *domain-independent ontology* with all relationships between classes of

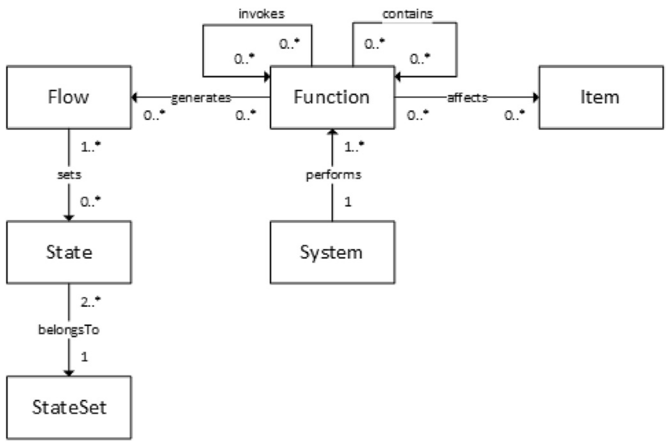


Fig. 1. Boilerplate attributes definition relationships in [9].

attributes in a system context (e.g. *function* invokes *function*, *system* performs *function*). Fig. 1 depicts the domain-independent ontology that we used in [9]. These relationships characterise every concept instance of the domain-specific ontologies in the requirements and provide information for building the formal model of the system [18] and for deriving verifiable properties [9,19–21]. They also set a *semantic analysis* framework, which is used to detect missing information and potential inconsistencies.

In [10,12,22], the following semantic analyses are proposed:

- *Missing information.* Requirements that are incomplete or still missing. In the former case, some references to concept instances are missing for one or more boilerplate attributes. In the latter case, we focus on boilerplate attributes that have been instantiated, but they have not yet been specified in any other requirement.
- *Conflicting requirements.* Similar requirements in terms of boilerplates that assign to the same attributes, for the same subject, different concept instances or use contradicting words or different quantities.
- *Underspecifications.* Requirements that assign attributes, which can be replaced with instances of more concrete subclasses.
- *Noise.* Requirements with terms not defined in domain ontologies.
- *Opacity.* Requirements which assign irrelevant instances of concepts to their attributes.
- *Redundancy.* Requirements with the same boilerplates and quantities that assign semantically equivalent concepts to their attributes.

These ontology-based analyses can be implemented using SPARQL queries [23] and SPIN inference rules [24].

Any omission detected by the mentioned analyses enacts a need of refining one or more requirements. The refinement of a set of requirements [25] consists of: (i) identifying what cannot be guaranteed or effected and (ii) augmenting or replacing them until they are fully verifiable. This can be ensured only after having derived formal properties that are really verifiable.

Example 1. This illustrative example was inspired from [26], where it was first used to demonstrate space system engineering methods for on-board software design. It refers to a generic first-in-first-out queue (FIFO), whose functionality is given in Fig. 2 as SysML natural language requirements linked with "refine" relations (directed arrows, where the target refines the source).

The "concurrent readers/writers" and "element types" requirements refer to the FIFO queue interface, whereas the "implementation language" requirement to an expectation, which by definition cannot be rendered verifiable. All requirements, apart from the "implementation language" were manually imported in our requirements ontology using a boilerplate syntax similar to the one in [9]. Fig. 3 demonstrates how the P15 prefix and M10 main boilerplates were combined for specifying the "Writer capability" requirement.

The process followed to input requirements into the ontology is shown in Fig. 4. First, the boilerplates to be used from the list of existing boilerplates are selected and their placeholders are presented according to the attributes of their syntax. Each placeholder is then filled in using either a (sub-)class of the boilerplate attributes or an instance of that (sub-)class (Fig. 3).

All requirements with a (sub-)class as an attribute are noted as having missing information. Complete requirements (i.e., with instances in each placeholder) are analyzed by applying the mentioned analyses that provide warnings in a table (e.g. which attribute has not been instantiated by any requirement, conflicting requirements etc). Warnings prompt to edit, refine or add additional requirements and then to repeat the semantic analysis in the updated requirements. In this example, the results of missing information analysis prompted us to define the "full" and "empty" FIFO states.

Semantic analysis is based on the ontological representation of the boilerplate attribute values in Fig. 5 (relationships comply with the domain-independent ontology of Fig. 1). Note that this representation refers to concepts that do not exist in the boilerplate representation of the requirement in Fig. 3 (e.g. `function_001`, `function_002`, `function_003`, `flow_001`). The reason is that the following two specification patterns are utilized.

First, in the main boilerplate, we refer to two different systems (i.e. the `writer` and the `FIFO`). This specification style, due to the used boilerplate, entails the invocation of two different functions; it is therefore represented with the `invokes` relationship between the `push` function (in main), and a new function, say `function_001`. For the sake of understandability, we name `function_001` as `add`, to express that

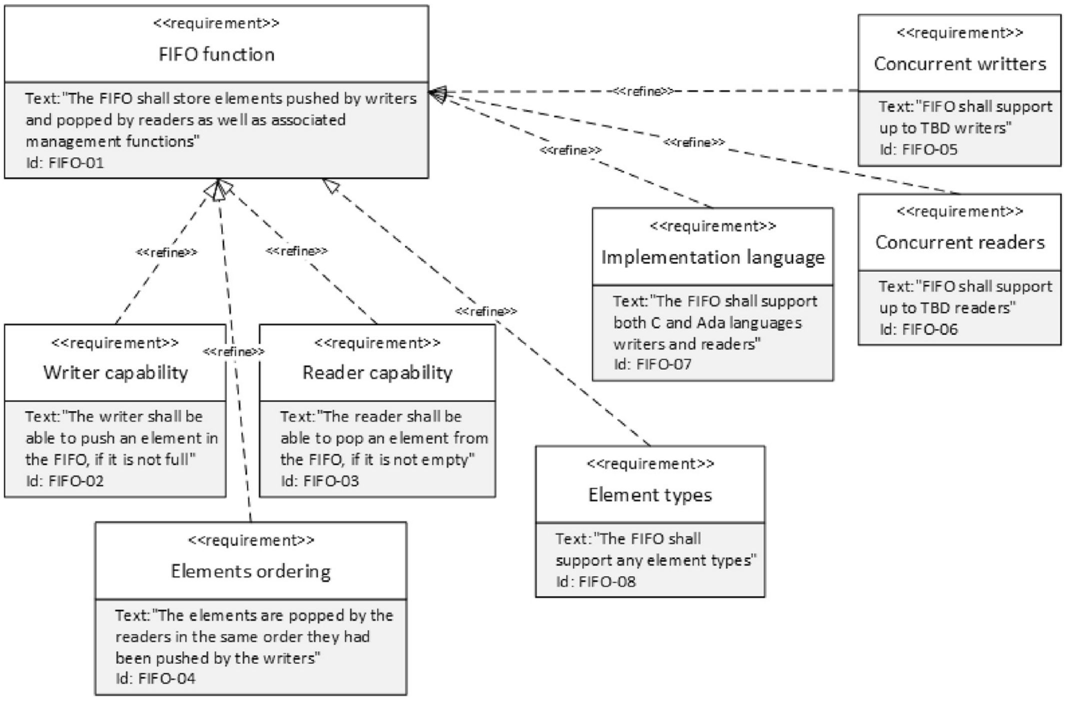

<<requirement>>
FIFO function
Text:"The FIFO shall store elements pushed by writers and popped by readers as well as associated management functions"
Id: FIFO-01
<<requirement>>
Concurrent writters
Text:"FIFO shall support up to TBD writers"
Id: FIFO-05
<<requirement>>
Implementation language
Text:"The FIFO shall support both C and Ada languages writers and readers"
Id: FIFO-07
<<requirement>>
Concurrent readers
Text:"FIFO shall support up to TBD readers"
Id: FIFO-06
<<requirement>>
Writer capability
Text:"The writer shall be able to push an element in the FIFO, if it is not full"
Id: FIFO-02
<<requirement>>
Reader capability
Text:"The reader shall be able to pop an element from the FIFO, if it is not empty"
Id: FIFO-03
<<requirement>>
Element types
Text:"The FIFO shall support any element types"
Id: FIFO-08
<<requirement>>
Elements ordering
Text:"The elements are popped by the readers in the same order they had been pushed by the writers"
Id: FIFO-04
<<refine>>

Fig. 2. FIFO requirements.

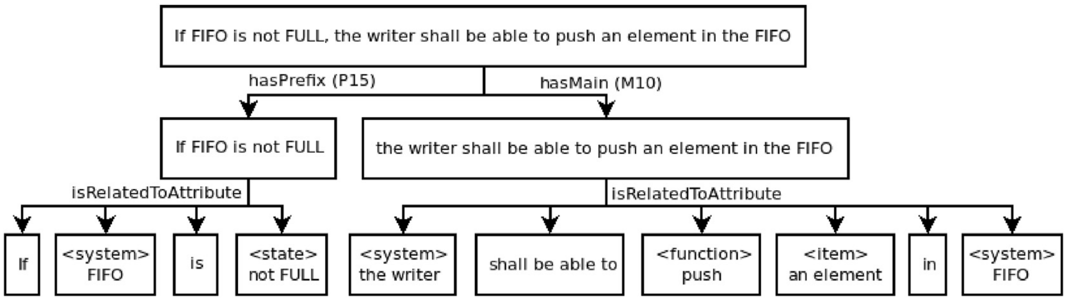


Fig. 3. Requirement boilerplate instantiation for the Writer capability requirement.

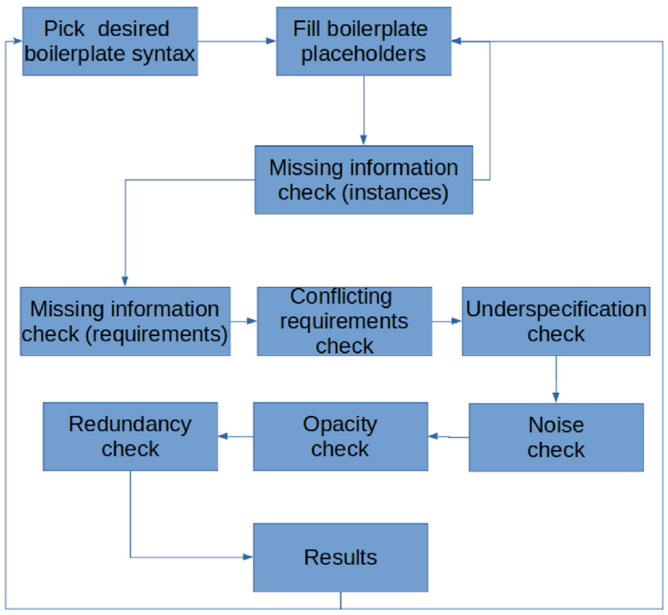

Pick desired boilerplate syntax
Fill boilerplate placeholders
Missing information check (instances)
Missing information check (requirements)
Conflicting requirements check
Underspecification check
Redundancy check
Opacity check
Noise check
Results

Fig. 4. Requirement specification process steps.

upon a `push` by the `writer`, the `FIFO` shall add an element to the queue.

The second specification pattern concerns with referring to a state as precondition specified in the prefix. This style, due to the prefix boilerplate, entails a representation according to Fig. 1, which assumes an additional flow that `sets` the state (and the `not` state) and two additional functions:

- `function_002` to check the state defined in the prefix boilerplate,
- `function_003` that acts as container, which contains the latter and the `function_001` that `FIFO` performs, as was previously mentioned.

Due to the container, relationships of `function_001` (including `invokes`) are applied to the container (`function_003`). We call `function_002` as `check` NOT FULL, and `function_003` as `check` NOT FULL and add. Fig. 5 eventually reflects the semantics: the `writer` performs a `push`, which `invokes` `FIFO` to check that state is not `FULL` and add element to the queue.

The described specification patterns can be applied by integrating appropriate SPARQL rules, in order to infer the additional functions and relationships. Alternatively, two or more separate requirements should be specified to define what each system performs and how their functions are invoked. △

The process of Fig. 4 integrates the semantic analyses of [10,12,22] together with those provided by our ontology architecture. Any variation of this process is feasible if all completeness checks precede the rest of semantic analysis. Completeness checks may even take place concurrently and interleave with the specification of requirements. From the discussion on the ontological representation of the requirement in Fig. 3, we see that completeness depends on (i) the relationships of boilerplate attributes (domain-independent ontology) and (ii) the boilerplates syntax and all concepts that are not explicitly referred as attribute values, but are inferred from them.

Example 2. This example refers to the Electrical Power Subsystem

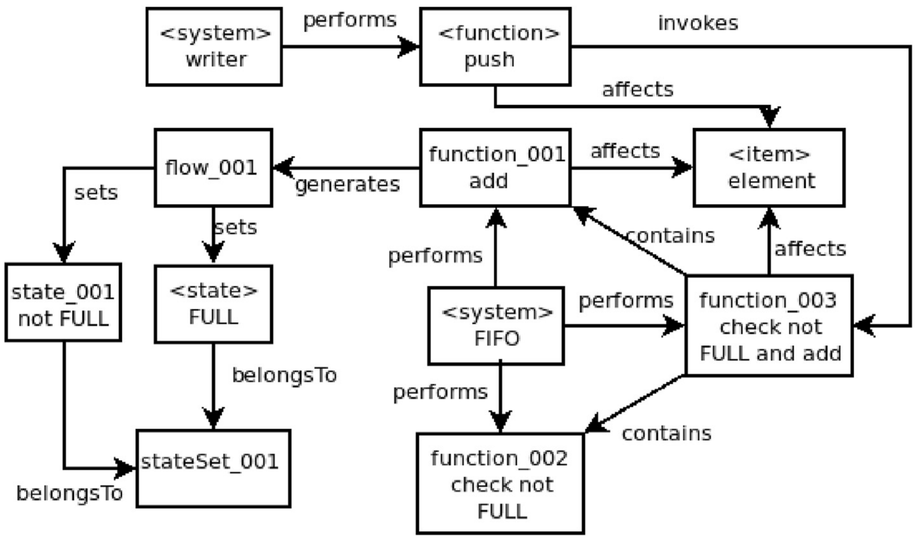


Fig. 5. Ontological representation of the Writer capability requirement.

(EPS) of a spacecraft system for generating, storing, conditioning and provision of electric power to all satellite units. EPS is decomposed into subunits (Fig. 6), whose requirements have been expressed using our boilerplate syntax (and attributes in Fig. 1) as shown in Table 1. The main subunits are:

- Power Conditioning and Distribution Unit (PCDU): receives the electric power from the solar array and/or battery and distributes it to the satellite subsystems through the Latching Current Limiter (LCL).
- Battery: stores electric power provided by the solar array during Sun visibility and powers the satellite subsystems during eclipse phases. It is charged/discharged by the Battery Charge/Discharge Regulator (BCDR).
- Main Error Amplifier (MEA): stabilizes power to the default voltage.
- Solar Array: delivers power to the satellite subsystems through the PCDU. It supplies the satellite during Sun exposure and in parallel charges the battery after the eclipse phases.
- PCDU Telemetry & Telecommand (TMTC): controls PCDU with commands external to the subsystem, e.g. power on/off, (dis-) charging from Attitude & Orbit Control System (AOCS) and ground control.

All requirements based on the M10 boilerplate (EPS-07, EPS-08, EPS-17 and EPS-18) resemble the requirement of Fig. 3, where the main clause refers to two different *system* attributes of the boilerplate. Similarly to Example 1, the main clause representation includes (EPS-17 requirement in Fig. 7) the `invokes` relationship between the function referred in the boilerplate and an additional function (shown as `function_001`), which (the analysis warns that) should be instantiated and specified in another requirement. We call `function_001` as `process`, in order to denote that ground control is expected to process telemetries upon receiving them.

Requirements that refer to a state in the prefix clause (EPS-10 to EPS-13), similarly to the requirement of Fig. 3, introduce a precondition to a function occurrence. This semantics requires two additional functions, as in Fig. 8, since there is no direct relationship in our domain-independent ontology (Fig. 1) between any state and function attributes:

- a function (`function_001`) to check the state in the prefix clause,
- a function (`function_002`) to act as a container of the latter and the function that is specified in the main clause.

If the state attribute in prefix refers to an item, that item is related to the new functions with the `affects` relationship.

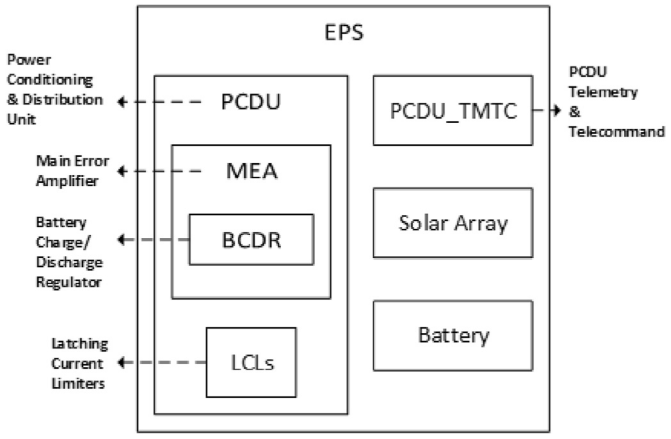
EPS-15 and EPS-16 use a different combination of prefix/main boilerplates that shows the need for an `invokes` relationship between function attributes. In this case, such a representation is not attributed to the different system attributes (AOCS or ground control and PCDU TMTC) in prefix and main, but it is based solely on the use of the function attribute in prefix. As shown in Fig. 9, the AOCS and ground control `perform` a `forwards` function, which `invokes` a `process` function performed by PCDU TMTC.

These specification cases show the role of prefix boilerplate in the `invokes` relationship definition of the requirement's ontological representation:

- when the prefix introduces a state attribute as precondition of main, additional functions and flows are needed apart from those that are explicitly mentioned in the requirement;

Table 1
Power subsystem requirements.

ID	Requirement
EPS-01	*M16: <System: EPS> shall contain a <System: PCDU>.*
EPS-02	*M16: <System: EPS> shall contain a <System: Solar Array>.*
EPS-03	*M16: <System: EPS> shall contain a <System: Battery>.*
EPS-04	*M16: <System: PCDU > shall contain a <System: MEA>.*
EPS-05	*M16: <System: MEA> shall contain a <System: BCDR>.*
EPS-06	*M12: <System: EPS> shall have < StateSet: operational modes> with values: <State: On>, <State: Off>.*
EPS-07	*M10: <System: EPS> shall < Function: transmit> <Item: bus voltage > to < System: platform>*
EPS-08	*M10: <System: EPS> shall < Function: transmit> <Item: bus voltage > to < System: payload>*
EPS-09	*M7: <System: EPS> shall < Function: start-up>, S2: after < Flow: launcher separates>.*
EPS-10	*P11: If < Item: bus voltage> is <State: undervoltage>, M7: <System: PCDU > shall < Function: power down certain LCLs>, S6: sequentially.*
EPS-11	*P11: If < Item: bus voltage> is <State: overvoltage>, M7: <System: MEA> shall < Function: reset> <Item: bus voltage > to < State: 50 V ± 0.5 V>.*
EPS-12	*P11: If < Item: battery voltage> is <State: below 45V>, M7: <System: BCDR > shall < Function: charge> <Item: battery voltage > to < State: 50V ± 0.5 V>.*
EPS-13	*P11: If < Item: battery voltage> is <State: over 55V>, M7: <System: BCDR > shall < Function: discharge> <Item: battery voltage > to < State: 50V ± 0.5 V>.*
EPS-14	*M7: <System: EPS> shall < Function: transmit> <Item: operational, health and performance telemetries of all units>.*
EPS-15	*P12: If < System: AOCS> <Function: forwards> <Item: EPS related telecommands>, M7: <System: PCDU TMTC > shall < Function: process> <Item: EPS related telecommands>.*
EPS-16	*P12: If < System: ground control> <Function: forwards> <Item: EPS related telecommands>, M7: <System: PCDU TMTC > shall < Function: process> <Item: EPS related telecommands>.*
EPS-17	*M10: <System: PCDU TMTC > shall < Function: transmit> <Item: telemetries > to < System: ground control>.*
EPS-18	*M10: <System: PCDU TMTC > shall < Function: transmit> <Item: telemetries > to < System: AOCS>.*


EPS
Power Conditioning & Distribution Unit
PCDU
Main Error Amplifier
MEA
Battery Charge/ Discharge Regulator
BCDR
Latching Current Limiters
LCLs
PCDU_TMTC
PCDU Telemetry & Telecommand
Solar Array
Battery
Fig. 6. Electric Power System decomposition.

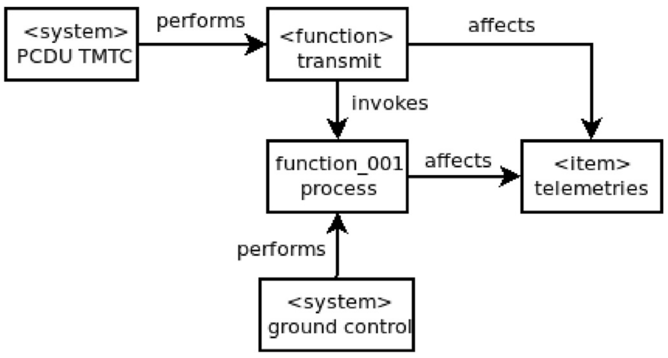


Fig. 7. Ontological representation of the EPS-17 requirement.

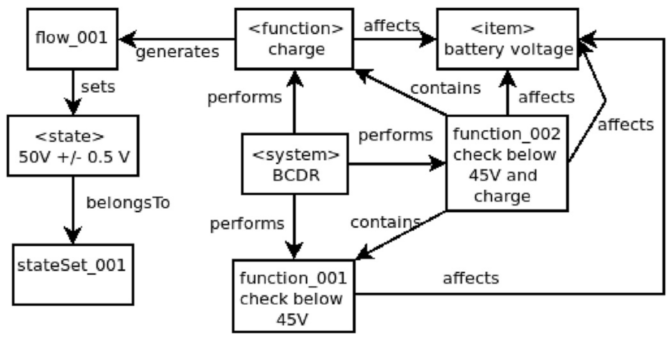


Fig. 8. Ontological representation of the EPS-12 requirement.

- when the prefix introduces a function attribute as precondition of main, an `invokes` relationship is generated without additional definitions.

Finally, we highlight the role of a class of requirements, found in requirements documents, which cannot be expressed ontologically with the concepts of a domain-independent ontology. Instead, such requirements complement the domain-specific ontologies. Specifically, EPS-01 to EPS-06 are related to the design of the specified system and if they are not handled properly, they will cause warnings for detected noise or opacity.

As noted, all specification patterns can be applied by integrating SPARQL rules. Upon semantic analysis, newly created concepts are marked as missing information, until they are further specified in other requirements. △

4. Formal properties derivation and verification

Even if we can guarantee absence of semantic omissions in requirements specification, this does not mean that requirements are valid for an acceptable system design. They could be unsatisfiable or not realizable! The conjunctions and prepositions in prefix/suffix patterns cannot be evaluated over a declarative-style specification, such as that created by instantiating boilerplates using domain ontologies. We need the model of a system design, in a *language with formally-defined operational semantics*. Then, from the various combinations of prefix, main and suffix clauses we derive formal properties referring to events and state variables of the model that correspond to the attribute values of the requirements. In *component-based* formalisms such as BIP [27] and the SLIM language (a subset of the Architecture Analysis & Design Language - AADL extended with behavioural models) of the COMPASS toolset [28], properties are specified in terms of atomic propositions that refer to specific components and their ports. Moreover, in SLIM, properties may also refer to error variables and states of error models [29,30].

Derivation of properties can be based on property patterns that are parameterizable, formalism-independent specification abstractions,

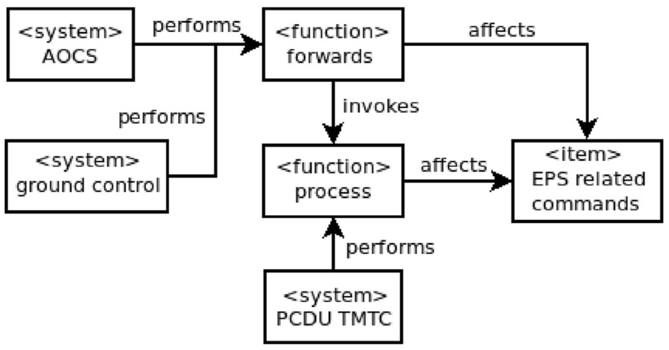


Fig. 9. Ontological representation of the EPS-15 and EPS-16 requirement.

which capture recurring problems in requirements formalisation. Patterns are used as a data input mechanism to specify properties; they are expressed based on a structured English grammar and they have formal semantics in logic languages. Each boilerplate is associated with a set of property patterns [9, 10], such that every possible requirement specification is covered by adequate properties. This is a prerequisite, for a logically consistent interpretation of requirements with respect to the model of system design.

Example 3. *An example of a properties derivation proposal that we introduced in* [9] *is the following. Let us consider the prefix boilerplate:*

P2: if <*Flow: e1*> and <*State: s1*>

When P2 is used in a requirement specification, two properties are derived that are expressed through the following property patterns from [31].

P2.1: globally, occ (*e1*) ∧ obs (*s1*) precedes *beg(M)*

P2.2: globally, *beg(M)* responds to occ (*e1*) ∧ obs (*s1*).

where *beg(M)*, *occ(e1)*, *obs(s1)* are mapped to events (component ports) corresponding to the occurrence of the main specification (*beg(M)*), the occurrence of *e1* and the observation of *s1* in execution traces of the model of system design. P2.1 is a safety property, whereas P2.2 is a liveness property; for both, a formal semantics has been defined in temporal logic languages. △

The complexity of the properties formalisation problem is discussed in [32], where the author points out that a sufficiently high level of precision must be retained. The author examines the "simple" requirement "When you pick up the phone, you get a dialtone" and provides four possible interpretations in Linear-time Temporal Logic (LTL) [33] with different levels of precision. It is eventually shown that for one who is not experienced in using temporal logic, it is unlikely to produce a precise formalisation of this requirement. Moreover, even when such a precise formalisation is achieved, it is impossible to recognize the initial specification, where the property comes from.

A widely used repository of patterns for functional properties is reported in [31]. The semantics of these patterns [34] is defined in Computation Tree Logic (CTL) [35] and in other logic languages. According to this approach, every property is specified using a scope (optional) and a property pattern. The scope (six different patterns) selects the subset of the model state-space, where the property is expected to hold; for the rest of the state-space, the property is undefined. The property pattern defines an expected occurrence of a given event/state (four different patterns) or the relative order of multiple events/states (four patterns) during system execution.

For extra-functional properties, the authors of [36,37] were based on the repository of [31], to introduce patterns for real-time specifications, while in [38] the authors organized them in a unified framework and introduced some additional patterns, which complete this repository. A different set of patterns for timing properties has been also proposed in [39]. Finally, in [40] the author has presented a repository of patterns for probabilistic properties, which have been used for real-world requirements focused on reliability, availability, performance, safety, security, and performability.

However, for verifying extra-functional properties we need a model of the system's design in a formal language with suitable semantics (e.g. timed automata for real-time properties, stochastic semantics for probabilistic properties). With respect to this, for the BIP language it is worth to mention:

- the BIP extension in [41] for specifying probabilistic aspects and a stochastic semantics for the parallel composition of BIP components;
- the RT-BIP extension for modeling timing constraints as a timed automaton and for computing schedules meeting these constraints [42].

COMPASS implements a different approach and supports various analyses. Instead of extending the SLIM language semantics, the tool utilizes model extensions, such as error models and fault injections, which enable it to automatically inject faults into the nominal model. The extended model is translated into a symbolic model amenable to formal verification and to a Markov chain for probabilistic analyses. A combination of backend tools supports a multitide of analyses [43], including properties verification, functional correctness, fault management, dependability and performability analyses.

4.1. Model-based and component-based system design

From the preceding discussion, derivation of formal properties from system requirements is tightly intertwined with the *system design*, i.e. the problem of defining the architecture, the modules, the interfaces and the data in a system model. A widespread approach is the component-based design, in which system design takes place by integrating blocks of functionality called components that are coordinated so as to fulfill the system requirements.

Two different perspectives of model-based and component-based system design exist, the so-called *top-down* and *bottom-up design* approaches.

In top-down design [44–47], we have a top-down system decomposition into components through which we mainly focus on allocating requirements to components, such that higher-level requirements are established. Each component is seen as an open system, with inputs provided by other components in the system or the external world and some outputs. All other components and the exterior world is referred to as the component's *environment*. Components are designed for a particular context and cannot constrain their environment. Requirements are formalised as *contracts* [48] consisting of *assumptions* for the environment and *guarantees* provided by the component. In [49], a pattern-based contract specification approach was introduced. When allocating requirements to a component [50], we must ensure that the assumptions for the component's environment (assertions on its input or invariants) can be fulfilled. An overly weak assumption on the environment can lead to overly strong (inconsistent) guarantees. Additional assumptions may be needed to weaken the existing requirements. The RATSY tool for property simulation can be used to improve the precision of additional assumptions [51]. In overall, at each step, a design decision has to be made towards finding an appropriate component (and contract) decomposition through which all higher-level requirements can be met. For this purpose, a formal *refinement relation* was proposed in [52].

In bottom-up design [9,53], at each step, a formal *architecture pattern* [54] is applied to a set of components of the system model in BIP. Architectures are reusable BIP models that characterise the interactions between a set of component types. In essence, they are operators restricting component behaviour for *enforcing* one or more properties. To apply them, certain properties are assumed for their operands, which can be verified locally by inspecting the components to which they are applied. If an assumed property is not satisfied, the component behaviour has to be refined. A related theory [55] provides a framework for architecture composition that guarantees preservation of properties established in a previous design step. In overall, this is an *incremental design* approach that aims to avoid a posteriori verification as much as possible (*correctness-by-construction*). The design philosophy lies on the fact that when specifying system requirements, a significant part of the specification comes from adapting requirements (and design solutions) found in previous projects. Architecture patterns provide the means to formally capture common solutions to recurring design problems in an abstract and reusable form. They drive the choice of component coordination, decomposition into components and behaviour transformation, but to be able to enforce a worthwhile set of properties we need a sufficiently developed library of architecture patterns. This incurs a non-negligible investment cost. In a related research work [56], the authors open prospects for defining architecture patterns, which enforce quantitative properties.

In overall, formalisation of system requirements is a two-stage process of interpreting them (i) based on a semantic model of the system domain (specification), and (ii) with respect to a model of the system's design (properties derivation). Due to the creative nature of this process, its obvious dependencies on two types of models and the form of the system design process (top-down vs. bottom-up design), human involvement cannot be vanished and therefore it can be only partially automated. In next section, we survey some noteworthy proposals of tool-support in recent bibliography.

5. Tools

At least three tools have been presented in related bibliography, but their availability beyond the projects in which they were developed is unclear. A fourth tool is available as part of the COMPASS toolset [28], which facilitates formal specification of requirements in the context of AADL models [57].

In [11], the authors presented a tool called DODT that was developed in the frame of the CESAR project [10]. DODT allows for projectional editing and typing of requirements based on a boilerplate syntax (Requirements Specification Language - RSL), an attribute ontology and a domain-specific ontology. The ontology-based validation checks in [22] have been implemented, which allow detecting contradictions by pairwise comparison of requirements, nouns that are not defined in the ontology and so on. Finally, a language of property patterns with formal semantics is supported. Although DODT does not implement an exact association of boilerplates with property patterns, in [10] the authors provide suggestions for patterns that could be suitable for capturing a given requirement.

The EARS-CTRL tool [58] was introduced to ensure by construction well-formedness of requirements written using the Easy Approach to Requirements Syntax (EARS) [13,14] templates. The tool checks whether a controller can be synthesized from a set of requirements. If a controller cannot be synthesized, it is possible that conflicting requirements exist. EARS-CTRL allows for projectional requirements' editing, based on a glossary defined for the domain of controller synthesis. Requirements are analyzed as LTL formulas. The analysis effectiveness depends on user-defined semantic information (e.g. simple predicates) for the given glossary. Finally, model synthesis is limited to a fragment of LTL that involves the universal path quantifier, the next-step operator and the weak until temporal operator [59].

The RERD tool [9] supports the requirements specification and properties derivation based on boilerplates and property patterns, as well as the incremental construction of systems for discharging the derived properties through the bottom-up design approach described in Section 4.1. While editing requirements, the user interacts with underlying ontologies by querying them and accessing their class instances for matching terms. In this way, the vocabulary of boilerplate attributes is restricted to terms that are identifiable within domain-specific ontologies and ontologies specific to the system under design. The boilerplate attributes relationships in Fig. 1 play a key-role in semantic analysis and property derivation, where each boilerplate has been associated with predefined property patterns that can formally capture it. For property enforcement, the user can choose among the available architecture patterns and parameterize them by selecting components from the incrementally built BIP model of the system. Then, the absence of deadlocks in the resulting BIP model has to be checked, which is proposed to take place using the D-Finder tool [60]. For the properties that cannot be enforced using the existing architecture patterns, the user will have to use external tools, such as the nuXmv model checker [61].

Regarding the COMPASS-based approach, in [57], the authors presented a different approach, according to which formal properties are not instantiated from patterns by replacing their placeholders with states/-events associated with boilerplate-specified attributes, but through assigning values to *attributes of the system model*. More specifically, a set of properties was introduced, which allow to associate values to specific AADL model elements. A taxonomy of formal properties was then defined, with each of them being expressed by the values associated to the corresponding AADL properties or else using certain structures in the

AADL model (e.g. subcomponents and port connections). The formal semantics of the system properties relies on the behavioural semantics of the SLIM language, whereas the logic used to define most of them is a variant of Metric Temporal Logic (MTL) [62]. While the taxonomy of formal properties is comparatively limited in expressiveness, it completely eliminates the need to choose a pattern and invent an instantiation. The formal properties can be analyzed for consistency among different *abstraction levels* of the system model or can be used as assumptions/guarantees of components for contract-based design. For the latter intent, COMPASS requires to further specify a *contract refinement*, which links a contract to the contracts of subcomponents. This allows to perform various analyses, such as the consistency or entailment for any subset of contract properties, to check whether the contract refinements are correct, as well as to tighten a contract such that the refinement still holds.

6. Limitations and challenges for industrial-scale projects

Effectiveness of ontology-based and pattern-based requirement specification depends on: (i) the quality of the domain-specific ontologies, and (ii) the expressiveness of pattern languages and their potential to eliminate ambiguity.

Regarding the former matter, for the design of proper domain-specific ontologies we need to involve requirement engineers, ontology engineers and domain experts, who are in charge of the system design. Such a process requires gathering knowledge from previous projects (e.g. requirement specification documents, architecture design documents), identifying the important concepts that are not specific to these projects and then classifying them into abstract categories (identifiable functionality, exchanged information between systems etc.). Categories must then be assigned to boilerplate attributes from a conceptual model, such as the one in Fig. 1, and their interrelationships must be ontologically defined. It is also required to capture tacit knowledge for the design problem, which is necessary to infer e.g. that certain events or data ranges that respect the boilerplate attributes model syntactically are not relevant semantically. Such knowledge may come for example from laws of physics or other system engineering aspects. From the ontology engineering perspective, it is necessary to adopt adequate qualitative (cohesion, adaptability etc.) and quantitative (coupling, computational efficiency etc.) criteria, which help to uncover errors and inefficiencies regarding the modeling complexity and size of the ontologies. A complete treatment of this problem is given in [63]. An additional challenge is how to organise the collaboration/responsibility of the different departments/stakeholders that may be involved in various ontologies, which in large system projects may include, for example, domain ontologies common for multiple stakeholders.

Expressiveness of pattern languages is important for being able to formalise all system requirements that need to be validated. To a large extent, the expressiveness depends on the set of connective words supported by the syntax of the pattern language and their meaning. A restricted set of connectives limits the language expressiveness, whereas a more extensive set of connectives may render impossible to completely avoid ambiguity in syntax and semantics. For enhanced expressiveness, it may be even necessary to assign multiple meanings to a connective word, which will have to be distinguished by the word's position within the sentence. In any case, the syntax of pattern languages will have to be designed with extensibility in mind.

The effort needed to apply any of the reviewed processes/tools in an industrial project and their scalability potential raise important challenges. Most of them have been applied to proof-of-concept case studies based on subsets of real requirements e.g. from space systems, but none has been applied to the complete set of requirements of a real industrial project. For such a project scale, the challenges to be addressed include: (i) to document the relationship between the ontology-based requirement specifications and the requirement baselines, as well as the ontologies lifecycle within a project and across projects, (ii) to improve usability through increasing the technology readiness level of the tools, especially when some properties cannot be verified and it is necessary to identify the relevant sub-model for the correction.

7. Conclusions and future research prospects

We surveyed the recent advances in formalisation and early validation of system requirements. A series of related industrial research projects has delivered valuable experience on pattern languages for various types of requirements and on the derivation of formal properties. Requirement specification can be based on domain ontologies, which are used to identify missing information and eliminate inconsistencies and cases of underspecification. The properties derivation depends on a formal model of the system, which may be incrementally built through a component-based design process. We reviewed two related design paradigms and we discussed the problem of formalising requirements within their context. Finally, we reviewed tools that were recently introduced, in order to address the mentioned problems.

The presentation of these developments allowed us to comment on their strengths and weaknesses. It was also evident that important limitations still have to be addressed, as well as open challenges associated with applying such an approach in industrial-scale projects. Apart from these prospects, open problems for future work exist on: (i) raising the level of automation for the formalisation and validation of requirements. and (ii) extending the applicability to more types of extra-fuctional requirements/properties.

References

[1] Berry DM. Ambiguity in natural language requirements documents. In: Paech B, Martell C, editors. Innovations for requirement analysis. From stakeholders' needs to formal designs. Springer; 2008. p. 1–7.
[2] Berry DM, Kamsties E. Ambiguity in requirements specification. Boston, MA: Springer US; 2004. p. 7–44.
[3] Lyons J. Linguistic semantics: an introduction. Cambridge University Press; 1995.
[4] Lyons J. Semantics, 1 & vol. 2. Cambridge: Cambridge University Press; 1977.
[5] Firesmith D. Are your requirements complete? J Object Technol 2005;4:27–44.
[6] Geierhos M, Bäumer FS. How to complete customer requirements. In: natural language processing and information systems. Cham: Springer; 2016. p. 37–47.
[7] Geierhos M, Bä¤umer FS. In: Guesswork? Resolving vagueness in user-generated software requirements. 1 ed. Cambridge Scholars Publishing; 2017. p. 65–108.
[8] Fabbrini F, Fusani M, Gnesi S, Lami G. An automatic quality evaluation for natural language requirements. In: Proc. Of the 7th int. Workshop on RE. Foundation for Software Quality (REFSQ 2001); 2001.
[9] Stachtiari E, Mavridou A, Katsaros P, Bliudze S, Sifakis J. Early validation of system requirements and design through correctness-by-construction. J Syst Software 2018;145:52–78.
[10] Rajan A, Wahl T. CESAR-cost-efficient methods and processes for safety-relevant embedded systems. Springer; 2013.
[11] Farfeleder S, Moser T, Krall A, Stålhane T, Omoronyia I, Zojer H. Ontology-driven guidance for requirements elicitation. In: The semanic web: research and applications. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011. p. 212–26.
[12] Mahmud N, Seceleanu C, Ljungkrantz O. Resa : an ontology-based requirement specification language tailored to automotive systems. In: 10th IEEE int. Symp. On industrial embedded systems; 2015. p. 1–10.
[13] Mavin A, Wilkinson P, Harwood A, Novak M. Easy approach to requirements syntax (ears). In: Proc. Of 17th IEEE int. Requirements engineering conf. IEEE Computer Society; 2009. p. 317–22.
[14] Mavin A, Wilkinson P. Big ears (the return of "easy approach to requirements engineering"). In: Proc. Of 2010 18th IEEE int. Requirements engineering conf. IEEE Computer Society; 2010. p. 277–82.
[15] Lin J, Fox MS, Bilgic T. A requirement ontology for engineering design. Concurr Eng 1996;4:279–91.
[16] Stalhane T, Omoronyia I, Reicenbach F. Ontology-guided requirements and safety analysis. 2010.
[17] Hennig C, Viehl A, Kämpgen B, Eisenmann H. Ontology-based design of space systems. In: The semantic web – ISWC 2016. Cham: Springer; 2016. p. 308–24.
[18] Wagner DA, Bennett MB, Karban R, Rouquette N, Jenkins S, Ingham M. An ontology for state analysis: formalizing the mapping to sysml. In: 2012 IEEE aerospace conf. IEEE; 2012. p. 1–16.
[19] Yan R, Cheng C-H, Chai Y. Formal consistency checking over specifications in natural languages. In: Proc. Of the 2015 design. Automation & Test in Europe Conf. & Exhibition, DATE '15; 2015. p. 1677–82.
[20] Böschen M, Bogusch R, Fraga A, Rudat C. Bridging the gap between natural language requirements and formal specifications. In: Joint proc. Of REFSQ-2016 workshops of 22nd int. Conf. On requirements engineering. Foundation for Software Quality (REFSQ 2016); 2016.

[21] Mahmud N, Seceleanu C, Ljungkrantz O. Specification and semantic analysis of embedded systems requirements: from description logic to temporal logic. In: Software engineering and formal methods. Cham: Springer; 2017. p. 332–48.
[22] Kaiya H, Saeki M. Ontology based requirements analysis: lightweight semantic processing approach. In: Proc. Of the 5th int. Conf. On quality software, QSIC '05. IEEE Computer Society; 2005. p. 223–30.
[23] SPARQL query language for RDF - W3C recommendation. 2008. https://www.w3.org/TR/rdf-sparql-query/. [Accessed 31 December 2019].
[24] SPARQL inferencing notation (SPIN). 2019. http://spinrdf.org. [Accessed 31 December 2019].
[25] Zave P, Jackson M. Four dark corners of requirements engineering. ACM Trans Software Eng Methodol 1997;6:1–30.
[26] Cordet astrium (astrium, SciSys), cordet TAS (TAS, UPD, P&P SW) and ESA; with contribution by intecs, GMVand soft-wcare, definition of A reference on board software architecture basic constituents, technical report TEC-SWE/09-248/AJ ESA ESTEC. 2009.
[27] Basu A, Bensalem B, Bozga M, Combaz J, Jaber M, Nguyen T-H, Sifakis J. Rigorous component-based system design using the bip framework. IEEE Software 2011;28:41–8.
[28] Bozzano M, Bruintjes H, Cimatti A, Katoen J-P, Noll T, Tonetta S. Compass 3.0. In: Tools and algorithms for the construction and analysis of systems (TACAS). Cham: Springer; 2019. p. 379–85.
[29] Mokos K, Katsaros P, Bassiliades N, Vassiliadis V, Perrotin M. Towards compositional safety analysis via semantic representation of component failure behaviour. In: Proc. Of 8th joint conf. On knowledge-based software engineering. IOS Press; 2008. p. 405–14.
[30] Mokos K, Meditskos G, Katsaros P, Bassiliades N, Vasiliades V. Ontology-based model driven engineering for safety verification. In: 36th EUROMICRO conf. On software engineering and advanced applications (SEAA 2010). IEEE Computer Society; 2010. p. 47–54.
[31] Dwyer MB, Avrunin GS, Corbett JC. Patterns in property specifications for finite-state verification. In: Proc. Of the 21st int. Conf. On software engineering, ICSE '99. ACM; 1999. p. 411–20.
[32] Holzmann GJ. Formalizing requirements is hard. Cham: Springer; 2019. p. 51–6.
[33] Pnueli A. The temporal logic of programs. In: Proc. Of the 18th annual symp. on foundations of computer science, SFCS '77. IEEE Computer Society; 1977. p. 46–57.
[34] Temporal specification patterns. 2019. http://patterns.projects.cs.ksu.edu/. [Accessed 31 December 2019].
[35] Clarke EM, Emerson EA. Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen D, editor. Logics of programs. Berlin, Heidelberg: Springer Berlin Heidelberg; 1982. p. 52–71.
[36] Konrad S, Cheng BHC. Real-time specification patterns. In: Proc. Of the 27th int. Conf. On software engineering. ICSE '05, ACM; 2005. p. 372–81.
[37] Abid N, Dal Zilio S, Le Botlan D. Real-time specification patterns and tools. In: Stoelinga M, Pinger R, editors. Formal methods for industrial critical systems. Springer Berlin Heidelberg; 2012. p. 1–15.
[38] Expressing and organizing real-time specification patterns via temporal logics. J Syst Software 2009;82:183–96.
[39] Reinkemeier P, Stierand I, Rehkop P, Henkler S. A pattern-based requirement specification language: mapping automotive specific timing requirements. In: Software engineering 2011 â€“ workshopband. Bonn: Gesellschaft fÃ¼r Informatik e.V.; 2011. p. 99–108.
[40] Grunske L. Specification patterns for probabilistic quality properties. In: Proc. Of the 30th int. Conf. On software engineering. ICSE '08, ACM; 2008. p. 31–40.
[41] Nouri A, Bensalem S, Bozga M, Delahaye B, Jegourel C, Legay A. Statistical model checking qos properties of systems with sbip. Int J Software Tool Technol Tran 2015;17:171–85.
[42] Abdelatif T, Combaz J, Sifakis J. Rigorous implementation of real-time systems from theory to application. Math Struct Comput Sci 2013;23:882–914.
[43] Bozzano M, Cimatti A, Katoen J-P, Katsaros P, Mokos K, Nguyen VY, Noll T, Postma B, Roveri M. Spacecraft early design validation using formal methods. Reliab Eng Syst Saf 2014;132:20–35.
[44] Murugesan A, Whalen MW, Rayadurgam S, Heimdahl MP. Compositional verification of a medical device system. ACM SIGAda - Ada Lett 2013;33:51–64.
[45] Murugesan A, Heimdahl MPE, Whalen MW, Rayadurgam S, Komp J, Duan L, Kim B-G, Sokolsky O, Lee I. From requirements to code: model based development of a medical cyber physical system. In: Software engineering in health care. Cham: Springer; 2017. p. 96–112.
[46] Whalen MW, Gacek A, Cofer D, Murugesan A, Heimdahl MPE, Rayadurgam S. Your "what" is my "how": iteration and hierarchy in system design. IEEE Software 2013;30:54–60.
[47] Bozzano M, Cimatti A, Fernandes Pires A, Jones D, Kimberly G, Petri T, Robinson R, Tonetta S. Formal design and safety analysis of air6110 wheel brake system. In: Proc. 27th int. Conf. Computer aided verification, Part I, LNCS 9206; 2015. p. 518–35.
[48] Benveniste A, Caillaud B, Nickovic D, Passerone R, Raclet J-B, Reinkemeier P, Sangiovanni-Vincentelli A, Damm W, Henzinger T, Larsen KG. Contracts for systems design: theory, research report RR-8759, inria rennes bretagne atlantique. INRIA; 2015.
[49] Damm W, Hungar H, Josko B, Peikenkamp T, Stierand I. Using contract-based component specifications for virtual integration testing and architecture design. In: 2011 design. Automation Test in Europe; 2011. p. 1–6.
[50] Ingham MD, Day J, Donahue K, Kadesch A, Kennedy A, Khan MO, Post E, Standley S. A model-based approach to engineering behavior of complex aerospace systems. In: Infotech@Aerospace 2012, garden grove, California, USA, june 19-21, 2012; 2012.
[51] Bloem R, Cimatti A, Greimel K, Hofferek G, Könighofer R, Roveri M, Schuppan V, Seeber R. Ratsy – a new requirements analysis tool with synthesis. In: Computer aided verification. Springer Berlin Heidelberg; 2010. p. 425–9.
[52] Cimatti A, Dorigatti M, Tonetta S. Ocra: a tool for checking the refinement of temporal contracts. In: 2013 28th IEEE/ACM int. Conf. On automated software engineering (ASE); 2013. p. 702–5.
[53] Sifakis J. Rigorous system design. Foundations and Trends in Electronic Design Automation 2013;6:293–362.
[54] Mavridou A, Stachtiari E, Bliudze S, Ivanov A, Katsaros P, Sifakis J. Architecture-based design: a satellite on-board software case study. In: Formal aspects of component software. Cham: Springer; 2017. p. 260–79.
[55] Attie P, Baranov E, Bliudze S, Jaber M, Sifakis J. A general framework for architecture composability. Formal Aspect Comput 2016;28:207–31.
[56] Paraponiari P, Rahonis G. On weighted configuration logics. In: Formal aspects of component software. Cham: Springer; 2017. p. 98–116.
[57] Bos V, Bruintjes H, Tonetta S. Catalogue of system and software properties. In: Computer safety, reliability, and security. Cham: Springer; 2016. p. 88–101.
[58] Lúcio L, Rahman S, Cheng C, Mavin A. Just formal enough? automated analysis of EARS requirements. NFM 2017 2017:427–34. NASA formal methods - 9th int. Symp.
[59] Cheng C, Hamza Y, Ruess H. Structural synthesis for GXW specifications. In: 28th int. Conf. Computer aided verification; 2016. p. 95–117.
[60] Bensalem S, Griesmayer A, Legay A, Nguyen T-H, Sifakis J, Yan R. D-Finder 2: towards efficient correctness of incremental design. In: Proc. Of third int. Conf. On NASA formal methods. Springer; 2011. p. 453–8.
[61] Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S. The nuXmv symbolic model checker. In: Computer aided verification, LNCS 8559. Springer; 2014. p. 334–42.
[62] Koymans R. Specifying real-time properties with metric temporal logic. R Time Syst 1990;2:255–99.
[63] Vrandecic D. Ontology evaluation. Ph.D. thesis. Karlsruhe Institute of Technology; 2010.