Requirements Engineering (2022) 27:1–30
https://doi.org/10.1007/s00766-021-00359-z

ORIGINAL ARTICLE

Formal requirements modeling for cyber-physical systems engineering: an integrated solution based on FORM-L and Modelica

Daniel Bouskela[3] · Alberto Falcone[1] · Alfredo Garro[1] · Audrey Jardin[3] · Martin Otter[2] · Nguyen Thuy[3] · Andrea Tundis[4]

Received: 17 July 2020 / Accepted: 17 July 2021 / Published online: 14 August 2021

Abstract
The increasing complexity of cyber-physical systems (CPSs) makes their design, development and operation extremely challenging. Due to the nature of CPS that involves many heterogeneous components, which are often designed and developed by organizations belonging to different engineering domains, it is difficult to manage, trace and verify their properties, requirements and constraints throughout their lifecycle by using classical techniques. In this context, the paper presents an integrated solution to formally define system requirements and automate their verification through simulation. The solution is based on the FOrmal Requirements Modeling Language and the Modelica language. The solution is exemplified through two case studies concerning a Trailing-Edge High-Lift system and a Heating, Ventilation and Air Conditioning system.

Keywords Modeling and simulation · Formal properties modeling · Requirements engineering · Modelica · System verification

1 Introduction

Cyber-physical systems (CPSs) are constantly increasing in complexity and sophistication, involving many components that are often designed and developed by organizations belonging to different engineering domains, including but not limited to mechanical, electrical, and software. Each component contributes to the functioning of the entire system, but in general, the behavior of the whole CPS cannot be straightforwardly derived from the behavior of its individual components [1, 2]. This increase in complexity makes the design, development, and operation of CPS extremely challenging. Furthermore, in order to have an optimal design, it is necessary to consider requirements along with operational constraints from the beginning of the design stage.

Requirement engineering (RE) is a major area of study in systems engineering with the purpose of discovering, developing, tracing, qualifying, communicating and managing requirements that define the system at successive levels of abstraction [3]. It involves five activities: (i) *discovering*,

✉ Alfredo Garro
alfredo.garro@dimes.unical.it

Daniel Bouskela
daniel.bouskela@edf.fr

Alberto Falcone
alberto.falcone@dimes.unical.it

Audrey Jardin
audrey.jardin@edf.fr

Martin Otter
martin.otter@dlr.de

Nguyen Thuy
n.thuy@edf.fr

Andrea Tundis
tundis@tk.tu-darmstadt.de

[1] Department of Informatics, Modeling, Electronics and Systems Engineering (DIMES), University of Calabria, via Ponte P. Bucci 41C, 87036 Rende, Italy

[2] Institute of System Dynamics and Control, German Aerospace Center (DLR), Oberpfaffenhofen Münchener Str. 20, D-82234 Wessling, Germany

[3] R&D Division, Electricité de France (EDF), quai Watier 6, Chatou Cedex 78401, France

[4] Department of Computer Science at Technische Universität Darmstadt, Hochschulstraße 10, D-64289 Darmstadt, Germany

which is devoted to gain knowledge on the project and identify the user requirements, objectives and other external constraints; (ii) *developing*, which focuses on delineating the functional and non-functional requirements into the requirements documents. Requirements must be documented so as to establish a requirements baseline with the stakeholders, start conceptualizing the system, and manage any changes; (iii) *tracing*, traceability of requirements to other artifacts, including requirements at other levels, provides a means to validate them against real-world needs, capture design rationale, and verify design against requirements; (iv) *qualifying*: it refers to all types of testing activities, covering design and solution testing, including unit, component, integration, system and acceptance tests (e.g., "verification" and "validation"); and, (v) *communicating and managing*: the collected requirements represent the baseline through which customers, suppliers, developers, users and regulators can agree on what they want to achieve.

The management of properties, requirements, and constraints in the area of utilization of CPS has increased dramatically since it involves many systems engineering aspects (e.g., functional and timing behaviors, performance, and reliability constraints) that need to be considered to ensure the proper operation of the CPS and reduce unexpected behaviors resulting from hazard and threats related to the interactions between the CPS and the environment in which it operates [4].

To address these challenges, new systems engineering methods and techniques are emerging to benefit from modeling and simulation (M and S). The main objectives are to support the functional validation of system requirements, design verification against requirements, testing, and verification of operational procedures. Moreover, functional and dysfunctional analyses can be supported in a better way through the simulation of different operational scenarios. Indeed, it is necessary not only to verify the nominal behavior of the system (e.g., the expected behavior of the system according to the offered functionalities) but to analyze how errors and faults can lead to system failures using suitable dysfunctional analysis techniques by combining M&S with traditional methods, such as PHA (preliminary hazard analysis) and FMEA (failure mode and effects analysis) [5–8].

Models provide consistent, complementary, and unambiguous representations used to formalize the structure and dynamics of a CPS and can also be exploited to investigate the effectiveness and consequences of design alternatives against requirements as well as to support system operation [9–11]. Verification is the confirmation process, through the provision of objective evidence that specified requirements have been satisfied; its purpose is to ascertain that each level of the implementation meets its specified requirements. To pursue these objectives, several research efforts are focusing their attention on the definition of integrated model-driven development processes, together with dedicated methodologies that can guarantee an objective checking of models [12, 13].

In this context, the paper identifies the main issues and challenges for the formal representation of requirements and proposes an integrated solution to overcome them. The solution permits the verification of system requirements through simulation by augmenting dynamic models defined with the *Modelica* language [14] with models that check requirements during simulation based on the *FOrmal Requirements Modeling Language (FORM-L)* [15]. The novelty of this paper is an integrated process of RE resulting from the integration and extension of the research results achieved within ITEA3 MODRIO (Model Driven Physical Systems Operation) (see, e.g., [16–19]), a European project that aimed at extending state-of-the-art modeling and simulation environments based on open standards to increase energy and transportation systems safety, dependability and performance throughout their lifecycle [10]. It is worth noting that the usefulness and interest in the presented results are shown by the use of its preliminary versions. As an example, in [20], a methodology, centered on the proposed property-based approach and the related Modelica library, previously released, is introduced and exemplified through a case study.

The rest of the paper is structured as follows. Section 2 provides an introduction to the essential concepts and background knowledge on the research domain and presents an overview of the main issues and challenges for properties modeling. For each identified issue and challenge, the existing literature contributions are presented in Sect. 3. Section 4 presents the proposed solution for simulation-based verification of requirements through a toolchain based on the *FORM-L* language. In Sect. 5, two Modelica libraries for the simulation of FORM-L requirements are presented. In Sect. 6, the proposed solution is exemplified by two case studies: the evaluation of different design variants of a Trailing-Edge High-Lift system, and the FMEA of an HVAC (Heating, Ventilation and Air Conditioning) system. Finally, conclusions and future work are delineated in Sect. 7.

2 Requirements modeling for cyber-physical systems engineering

2.1 Basic definitions

This section provides a brief overview of the essential definitions used throughout the paper.

In [21], cyber-physical systems (CPSs) are defined as *"smart systems that include engineered interacting networks of physical and computational components. These highly interconnected and integrated systems provide new functionalities to improve quality of life and enable technological*

advances in critical areas, such as personalized health care, emergency response, traffic flow management, smart manufacturing, defense and homeland security, and energy supply and use". Thus, CPS integrates the *physical* components (tangible physical devices) with the *cyber* subsystems (computational and communicational capabilities) to pursue specific objectives. CPSs are characterized by five aspects [21]: (i) *Reactive Computation*, unlike the traditional computational devices that produce an output when supplied with an input, CPSs are reactive systems where their components constantly interact with the environment. CPSs react to events, coming from sensors and actuators, by changing their behavior according to the events; (ii) *Concurrency*, CPS's components are running concurrently, exchanging data with one another to achieve a specific goal; (iii) *Feedback Control*, CPSs interact in an ongoing way with the environment in a feedback loop through controllers by gathering external information (e.g., pressure, temperature, and humidity); (iv) *Real-Time Computation*, real-time performance is a critical aspect since the correctness of CPS behaviors depends not only on the computational results but also on the physical time instant at which these results are produced; and (v) *Safety-Critical*, the design and implementation of CPS require a high level of assurance in their nominal behaviors because possible failures and errors can lead to unacceptable consequences that may cause injury or death to operators and damages to the environment. Examples of CPS are an Air Transportation System, an Attitude Determination and Control System, an On-Board Communication System [6, 7, 22].

CPS can be modeled and simulated as hybrid systems whose dynamics are regulated through a mix of continuous and discrete behaviors. Such systems evolve over time and can jump to an operation mode during which state variables are atomically updated. Generally, the continuous behavior is described through ordinary differential equations (ODE) or differential algebraic equations (DAE), whereas the discrete behavior is defined by a control graph. The state of CPS is defined by the values of their continuous variables in a given discrete mode. Continuous flow is allowed as long as invariants hold, whereas discrete transitions occur when a jump condition happens. Thus, an invariant is a property that holds in all reachable states of the system [23].

In this context, according to [12, 15], a *system property* can be defined as an expression that specifies a condition that must hold true at given times and places. System properties can be regarded as *requirements*, *assumptions*, and *guards*. *Requirements* are attributes, conditions, or capabilities that must be met or possessed by either a system or a component to satisfy a contract, standard, specification, or other formally imposed documents. *Assumptions* are properties that are supposed to be satisfied (e.g., simulation scenario assumes/ensures that are satisfied). *Guards* are conditions that must be satisfied for a system to be valid: a violation of a *guard* can signal that the physical behavior of the system (i.e., of the model representing and simulating it) is no more consistent, e.g., a condition deriving from a physical law is violated. Listings 1, 2, and 3 provide examples of a requirement (`required property`), an assumption (`assumed property`) and a guard (`guard property`) of a main power supply (MPS), respectively (as extracted from [15]). In FORM-L the keywords `requirement`, `assumption`, and `guard` have been also introduced to compactly indicate these concepts (see Sect. 4.1).

```
//The requirement indicates that the MPS must be
    deactivated when it goes under maintenance.
required property R2 =
 when MPS.state becomes maintenance
 check Active becomes false;
```

Listing 1 A required property of a MPS.

```
//The assumption states that when the MPS is not
    available, the operator will not launch a maintenance
    session (no eMaint).
assumed property A1 =
 during (MPS.state==Unavailable)
 check no eMaint;
```

Listing 2 An assumed property of a MPS concerning human operator's actions.

```
//When violated the guard indicates that the system is no
     more in a valid state as the internal resistance of a
     MPS must not be negative.
guard property G1 =
 during (MPS.state==Active)
 check MPS.InternalResistance>=0;
```

Listing 3 A guard property of a MPS.

System requirements are defined to ensure the proper operation of complex physical systems (such as power plants, aircraft, or vehicles), but also to state functionality that satisfies customer needs. The gathered requirements have to be validated and verified to guarantee that they meet the overall objective of the system and stakeholder needs. Requirements validation and verification involves evaluation, analysis, and testing to guarantee that a system satisfies its requirements. Specifically, the ISO/IEC/IEEE 15288 standard states *"Verification is the process for determining whether or not a product fulfills the requirements or specifications established for it". "Validation is the assessment of a planned or delivered system to meet the sponsor's operational need in the most realistic environment achievable"* [24].

2.2 Issues and challenges

The definition of suitable methodologies and techniques to support the realization of physical and computational components have been central themes in the scientific community in the last decades. CPS are complex and multidimensional systems that constantly interact with their environment and react to events coming from sensors and actuators, changing their behavior according to the type of events. The CPS components are running concurrently in real time which represents a critical aspect since the correctness of the CPS behavior depends not only on the computational results but also on the physical time instant at which these results are produced. The design and implementation of a CPS require a high-level of assurance of their nominal behavior because possible failures and errors can lead to unacceptable consequences that may cause injury or death to operators and damages to the environment [25]. In this context, different research directions and ideas towards requirements management have emerged, such as those based on the advancements of machine learning (ML) techniques [26]. However, in order to manage the requirements of CPS, a systematic approach is indeed needed, starting from the identification and clarification of the fundamental research challenges. The main issues and research challenges can be clustered into the following three main groups [17]:

1. *Conceptual properties representation*, which aims at focusing on how to express properties through models and identify the necessary concepts to capture the system characteristics in order to provide/create explicit and formal information, starting from implicit and informal inputs, which may be imprecise and ambiguous. Therefore, there is the need to understand how to identify system properties and what characteristics must be considered;
2. *Binding and automated model composition*, which deals with connecting models and provides mechanisms for enabling their composition. With reference to properties modeling, the main problem is how to bind the property model of a system (i.e., the definition of the system properties and their relationships) to its structural and behavioral models;
3. *Tracing and verification*, which aims at monitoring specific properties as well as exploiting simulation techniques for supporting and verifying their fulfillments. It refers to the ability to: (i) describe and follow the life of a property, in both forward and backward directions, through the whole system life cycle; (ii) exploit simulation techniques by benefitting from modern model-driven simulation tools for automating the verification of system requirements in an integrated framework. Tracing and verification are two sides of the same coin and an important part of a system development process.

Furthermore, the need for a full-fledged mechanism to support the system development process from the early design up to its operation allowing continuous integration in an automatic way (due to the high number of involved system components), is still an unsolved problem as highlighted in [27]. Here, the high-level idea of using a graph-based metadata approach by storing the data in a database, in order to support the automatic generation of execution orders and how to deploy participants and connect them together, is briefly introduced; the authors well highlight the need of having innovative tools to support it.

3 Related work

This section reports and discusses previous works and available solutions related to the research challenges identified in Subsection 2.2.

Addressing the issues introduced in Subsection 2.2 is not a trivial task to accomplish; indeed, it requires considerable research efforts as well as the employment and the cooperation of human resources with deep and wide knowledge and skills both in academic and industrial contexts, in different engineering fields and application domains. In the following, for each of the identified challenges (*conceptual properties representation*, *binding and automated model composition*, *tracing and verification*) the main available solutions are described.

3.1 Conceptual properties representation

As discussed in the previous section, conceptual properties representation refers to the first aspect that needs to be faced in the RE management field. In fact, requirements engineering means to provide a formal representation of requirements, which allows their computation, i.e., machine readable and automatically processable, and their evaluation without ambiguity, i.e., without misinterpretation. Basically, it refers to the definition of models based on the identification of specific features that can be used to characterize particular requirements. In this subsection, previous research contributions have been identified, which address the modeling and formal representation of requirements, with particular attention to modeling languages and approaches, as well as available software libraries that allow the requirements integration and evaluation via software, by highlighting for each of them pros, cons and their limitations.

In [28], a method for requirements modeling driven by pattern analysis is proposed. It is a conceptual process that aims to support data analysts by reusing previous experiences for the elicitation, modeling and analysis of requirements. The paper describes a high-level approach for supporting the modeling process which in turn relies on a strong human intervention, based on the feedback from analysts to adjust the modeled system behavior.

Another research effort regarding properties representation was conducted in the context of medical devices [29]. In particular, the authors discussed the need of having automated management of requirements by focusing on a system, called total knee arthroplasty (TKA). As first effort, they identified 43 properties, and among them 15 that should be fulfilled by their traces, were selected as representative. The paper lists these 15 properties and performs their classification. Furthermore, it identifies several stages in the product lifecycle where these properties should be evaluated. However, a formal definition of such properties is not provided in the paper.

In [30], a dataset, called PURE, of natural language requirements documents collected from the Web is presented, by defining a restricted vocabulary of domain-specific acronyms, words and sentences, and by highlighting some peculiarities in requirements terminology. However, as stated by the authors other important tasks such as requirements categorization, requirements traceability, ambiguity detection and equivalent requirements identification require a deeper investigation.

In [31], a system of systems (SoS) requirement development process model is proposed along with a toolkit which supports the modeling and analysis of SoSs requirement development. However, the proposal does not provide any information about how to model requirements in a formal way in terms of features and relationships between them. Moreover, the adoption of model-based approaches is very common, and typically two main kinds of trend to deal with conceptual properties representation exist in the literature: one based on the definition of user libraries, and the other on the exploitation of specific modeling languages.

In [19], a library-based approach is employed to deal with the properties representation. Its main benefit relies on being reusable and not affecting the tool-architecture; however, it is limited in terms of functions, as it is possible to model only pre-defined properties. An example of this approach is described in [12], where a set of predefined specific-domain properties centered on a threshold mechanism are presented. As a consequence, the modeling of new aspects and their related properties implies to extend the library.

The modeling of properties related to responsibilities is instead presented in [32]. The authors distinguished among them two different responsibilities' concepts: "safety" and "liveness". On the basis of such concepts, specific properties have been defined as rules and conceived as invariants.

Another way of approaching properties modeling is presented in [33, 34], which consists in extending the programming and/or the modeling language by introducing new keywords natively related to properties modeling concepts.

An example of such an approach is presented in [34], where native extensions of the Modelica language [35, 36] have been investigated, which have been integrated in it by updating the related Modelica compiler, in order to support the compilation of the new constructs, i.e., new "keywords" introduced at the language level. The main advantage of this approach relies on being very flexible and allows defining new properties, by using the new language extensions, i.e., the new embedded constructs, regardless of the specific application context. However, the disadvantage lies on the re-implementation of the Modelica compiler together with possible backward compatibility problems.

Another previous important research contribution for expressing and verifying requirements based on a temporal logic is *SALT (Smart Assertion Language for Temporal Logic)* [37], a specification language which follows a software engineering approach based on assertions. The use of approaches based on assertions was initially investigated and evaluated by some of the authors of this paper, whose results are presented in [34]. They conclude that this proposal is not useful due to the following problems : (i) highly invasive, since adding assertions means modifying/updating the code in existing software components (which is unthinkable), thus going against the principle of decoupling the system model from the requirements model, (ii) need for strong integration and dependencies, since introducing an external language in the Modelica language requires modifying the Modelica specification and introducing external dependencies.

A further approach is centered on the graphical representation of both systems and properties by using symbols for representing the concepts and diagrams (e.g., flow charts) to model its behavior at different representation levels. SysML is a specific profile of UML for modeling systems [38]. This means that SysML is an extended subset of UML, that is, some UML concepts are inherited and used in SysML as they are, others are customized, others are excluded and others are introduced from scratch. It provides a set of concepts for expressing constraints as well as requirement allocation, requirement satisfaction, requirement decomposition, and requirement derivation.

It is worth noting that existing languages such as *UML/SysML* or *AADL (Architecture Analysis & Design Language)* are general purpose. If, on the one hand, the use of such tools is widely applied at the system modeling level; on the other hand, the passage from the conceptual model to the realization model requires in-depth knowledge on both sides. Other research efforts in the literature tried to combine these languages such as in [39, 40], so as to obtain an integrated modeling tool. However, the main problem remains how to move from such model-based representations to executable versions. This is complicated when the logic behind modeling languages, such as GORE [41] which is goal oriented, is different from the one on which the Modelica implementation language is based, which is basically declarative and equation based.

KAOS is another example of a Goal-oriented language [42]. Although it provides linear temporal logic (LTL) features, unfortunately they are not *"acausal"* (e.g., the definition of models is done in a declarative style and the distinction between inputs and outputs do not affect the model and is only considered at run-time), but the use of the KAOS language requires specifying in advance what are the inputs and outputs.

That is why a specific solution to concretely reduce this gap among design and implementation is needed.

3.2 Binding and automated model composition

Binding and automated model composition are quite strictly interconnected issues, and many approaches have been proposed in the systems engineering field. In this subsection, research works centered on approaches for enabling automatic model compositions have been investigated, by mainly focusing on mechanisms which enables the binding among a property model to its structural and behavioral models.

For example, in [43], a tool called IoT Composer is presented, which supports the development of Internet-of-Things (IoT) applications by providing a behavioral model for objects and their binding and composition. In particular, the user can manually select some objects, and can graphically define a set of bindings between their interfaces to generate a composition (e.g., selected objects plus bindings).

In [44], a Petri net program generator is described which is used for the automated composition of cellular grid models. It is centered on some basic parameters such as the cellular grid size, the buffer capacity of the communication device, the number of packets in each device buffer for the automatic creation of hexagonal structures, which can be used in a wide range of Petri nets applications, including telecommunications.

In [45], the concept of multi-functionality and development of a holistic service composition model, centered on modeling and composition reasoning techniques is proposed. This means, techniques and algorithms that given a set of models, allow identifying and composing a subset of them automatically, in order to obtain a more complex model that satisfies the specified requirements. Additionally, since most existing composition reasoning techniques can only handle a single functionality, new algorithms for automated holistic service composition are provided. They extend existing models to enable more comprehensive composition requirement specifications, considering systems with multiple functionalities, offering users more choices, and handling exceptions.

The work presented in [46] aimed at improving the reusability of model transformations by automating their composition. In particular, the authors proposed an automated model transformation chain by formalizing models and transformations using alloy. However, with the increasing number of elements in the input model, the approach was not feasible because of the exponential growth of computational steps to identify the necessary model transformations. As a consequence, a further improvement has been proposed by introducing optimizations for the identification in the transformation chain by improving its scalability.

Furthermore, previous works with a particular focus on tools for automation of service composition are collected and shortly summarized in [47]. Some other important approaches are reported by distinguishing bindings at the

component level, service level, process level, and software level. Specifically:

- In [48] a research effort centers on a Client-Provider-based approach which relies on the concept of data, customer and provider. In particular, the data represent the point of contact between a client and a provider. The concept of client represents entities that use a particular data, whereas providers are entities that generate it. Clients and Providers can be in a many-to-many relation and they might not know each other directly. To deal with it a third entity is introduced, called Mediator, that manages and facilitates connections between clients and providers, and provides automatic assistance in order to avoid modeling errors and reduce the manual modeling effort for integrating models.
- An automated model composition, called Roman model, which is centered on the behavior of the services, is proposed in [49]. The composition task operates at the service level. It consists of synthesizing an orchestrator that preserves specified interactions among artifacts expressed as virtual target services in order to compose the overall model automatically.
- In [50] the authors proposed a feature-based automated model composition. It aims to identify and bind specific software modules on the basis of their exposed functionalities. In particular, specific adaptation rules, which are described in terms of features, are pre-configured to be able to separate the application logic from runtime adaptation mechanism.

3.3 Tracing and verification

Concerning the possibility of establishing traceability between models as well as to allow property verification, there are some popular approaches available in the literature mainly focused on requirements, which are discussed in the following. As a consequence, the research contributions discussed in these subsections have been selected by focusing not only on requirements tracing techniques but also on related available verification approaches.

For example, in [51], a research prototype for the validation of requirements at the model level is announced. It is supposed to support the validation of conceptual schemas by using testing obtained by implementing techniques for transforming instantiations from a requirements model into test cases. The general idea of the tool is presented. However, details and experiments are omitted leading to a lack of evidence.

In [52], a framework, called Sophia, is proposed that supports assurance of critical cyber physical systems using compositional model-based approaches. It helps to trace the developed analysis outcomes to the requirements in standards for compliance support by integrating models for supporting Process Hazard Analysis (PHA), FMEA, and Fault Tree Analysis (FTA) at the design stage.

The STIMULUS software from Dassault Systèmes allows system architects, at the specification phase, to define and verify requirements by using language templates, state machines, and block diagrams in order to detect ambiguous, incorrect, or missing requirements before the design phase. At the validation phase, STIMULUS also offers functionalities to support software testing to check that the source code is compliant with the system's specifications [53]. Other works are present in the literature aimed at supporting the system verification phase, such as the contribution described in [54], where a framework centered on a 3-value temporal-logic for system health management of real-time systems is presented, which uses a statistical assessment approach based on Bayesian network.

In [55], a method for continuous usage of scenarios, embedded in the systems engineering process, was proposed by dividing complex and intangible development goals into smaller solvable tasks. The authors place the tracing and verification task of the requirements in the step "Scenario-based methods" of the overall method. However, the details about how tracing and verification are performed are completely omitted, so as to make the experiment not reproducible.

In [56], the authors present a tool for generating monitors and a Simulink library for supporting model validation at simulation time. The main weakness of such work, as it is stated by the authors, is that no support to the user to describe the model requirements in a formal language with a syntax closer to the natural languages has been provided.

A recent work is presented in [57], where issues and challenges related to the capture and analysis of CPS requirements are highlighted, as CPS models typically involve time-varying and real-valued variables, and dynamic behaviors. The authors present the application of NASA Ames tools to perform end-to-end analysis of the Ten Lockheed Martin Challenge Problems (LMCPS), which are a set of industrial Matlab/Simulink model benchmarks and natural language requirements developed by domain experts to: (i) elicit and formalize the semantics of requirements gathered in natural language; (ii) generate Matlab/Simulink verification code; and (iii) perform their verification through model checkers. This is a further demonstration of the increasing interest by industries on these research topics.

In [58], traceability in Systems Engineering is discussed. In particular, the authors describe how to capture traceability information in a system with heterogeneous artifacts, by contextually presenting a case study in Avionics, which uses a traceability model and a reference trace links taxonomy. The model is presented in UML and well illustrated in the paper. However, it is unclear from the paper how the experimentation in Simulink is actually realized and tested.

Furthermore, the ability to perform requirements tracing can be accomplished by four different types of links, utilizing both forward and backward direction tracing approaches. The different links that can be created for requirement tracing are classified into four mechanisms, according to [59]: (i) *Forward from the requirements*, when a requirement is assigned to one or more system components that are responsible for the requirement. This approach allows the evaluation of the impact due to requirement change; (ii) *Backward to the requirements*, which allows to test and verify a requirement against a system or part of it by mapping back its compliance rules as a requirement; (iii) *Forward to the requirements*, that is adopted to map customers' needs and technical specifications to the requirements in order to evaluate the impact on the system when a change occurs; (iv) *Backward from the requirements*, which is typically adopted when the validation of requirements against the customers' wishes as well as technical assumptions is required.

Concerning formal verification, there are different techniques. Two major methods are model checking and theorem proving [60]. In particular, model checking is a technique that relies on building a finite state model of a system and checking that the desired property holds in that model. It is used in hardware and protocol verification. By using theorem proving, the system under consideration is modeled as a set of mathematical definitions using some formal mathematical logic, whereas the desired properties of the system are then derived as theorems that follow from these definitions. Other approaches are: (i) *state space enumeration*, which is based on the reachability of a state in order to check whether a system complies with a specific property. It is a method that works for a small and medium-size level of complexity, due to the potential exponential growth of the state space when the size of the system increases as well as related functionalities; (ii) *partial order reduction*, which tackles the problem of the exponential growth of states, by trying to select and use only the necessary and relevant ones to support the evaluation of the property under analysis; (iii) *symbolic model/checking*, which is a symbolic verification method centered on a symbolic representation of the transition relations by using Boolean expressions modeled as binary decision diagrams as well as convex polyhedrons to represent linear constraints. It uses the computation of a fixed point over an encoding of the state transition relation to determine the reachability of a given state.

An approach that combines the four mechanisms previously described, i.e., *Forward from the requirements*, *Backward to the requirements*, *Forward to the requirements*, and *Backward from the requirements*, is presented in [33], where a simulation-driven solution is centered on an equation-based style by employing concepts and relationships identified and described through a proposed meta-model in [34]. According to such proposal, a requirement can be evaluated to three possible states: (i) *Violated*, meaning that under the current simulation parameters a violation of the requirement was found; (ii) *Not violated*, meaning that under the current simulation parameters no violation of the requirement was found; (iii) *Not evaluated*, meaning that in the given scenario, the preconditions of the requirement were never fulfilled, and thus, the requirement was never evaluated. Once requirements are evaluated, the fulfilling relationship is computed according to a specific algorithm that aims to indicate: (1) when a property is violated, and (2) how the component responsible for a property violation can be identified. In particular, a property or a requirement can be violated, not only when a single component fails, but also when the interactions among two or more components are wrong. Basically, the verification and traceability process requires tracing all the fulfilling relationships for a given requirement, to reach a set of components that the requirement depends on. This set can be then analyzed, to detect that (i) either a component is not properly working, or (ii) the interaction among a set of components is not consistent, contrary to what was expected. In fact, such an approach distinguishes between two types of properties: (a) one that defines the expected behavior of a single Physical System Component; (b) the other that defines the expected behavior of two or more Physical System Components in terms of interactions between them. As a consequence, it is possible to either identify a single component as possibly liable or a set of components within which one or more components are responsible for violating the requirement.

3.4 Remarks

From the analysis conducted, it appears that the aspects discussed in the three previous subsections are of great importance as shown by the various research efforts conducted so far.

Unfortunately, the aspect of representation and modeling is typically faced in an isolated way without taking into account the other aspects, in fact, a lack of integrated tools clearly emerges; moreover, many of the solutions are either strictly linked to specific application context or too high-level, so as to make the conceptual model difficult to be transformed for its use in operation neither in simulation. With regard to the binding aspect, on the one hand, there are solutions that deal with it in a manual way, which makes it inapplicable in contexts with thousands of components; in other approaches, they use static criteria based on specific metadata, making these approaches not efficiently reusable, whereas concerning the tracing and verification mechanisms they have not been considered in the context of properties modeling as validation tools to assess the correctness of the system under specific conditions as well as

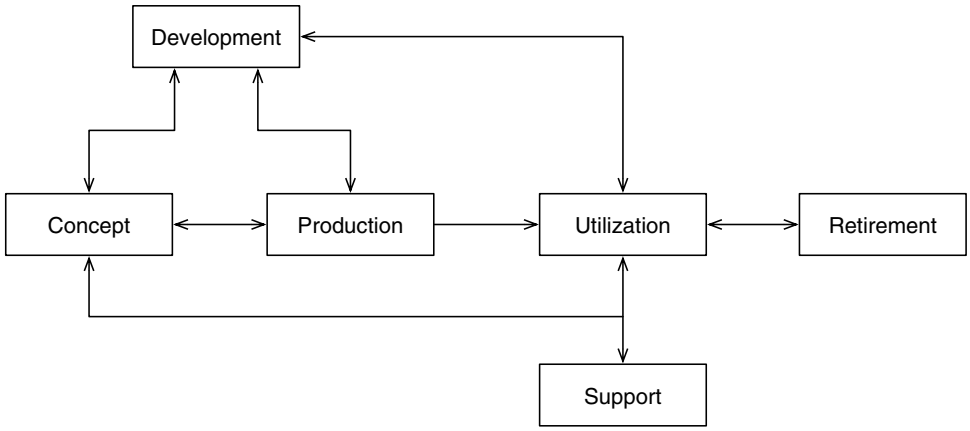

Fig. 1 The IEEE Std 24748-1-2011 System Life Cycle Stages

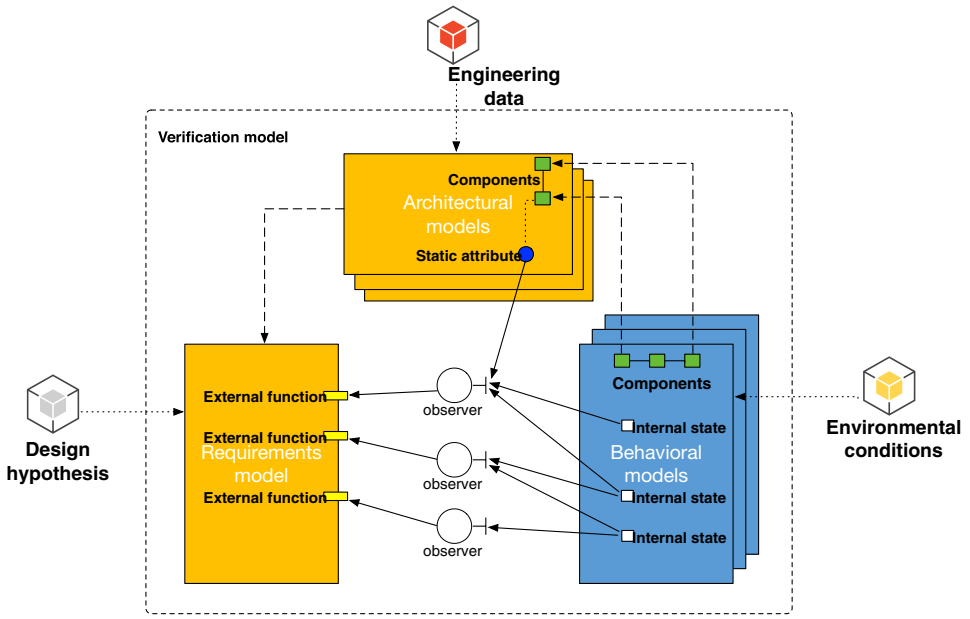

Fig. 2 The simulation-based verification of requirements solution

to identify anomalies resulting from component interactions or external factors.

Other standardization efforts are underway regarding possible approaches to support requirements management, such as the *OSLC (Open Services for Lifecycle Collaboration)* initiative that deals with specifications based on W3C *Resource Description Framework (RDF)*, Linked Data and REST, in order to allow integration at the data level through links between related resources [61]. This possibility, which envisaged extending the Modelica standard on the basis of other specifications such as those issued to OSLC, was taken into consideration, proposed and discussed within the Modeling Consortium. From the discussion emerged the will of not wanting to modify the Modelica standard in order to keep Modelica a pure and non-hybrid language for reasons of efficiency and compatibility with the currently available programming environments.

The proposed solution is based on the representation of requirements in a temporal-logic based language called FORM-L.

The idea behind FORM-L is to provide a compact and "distilled" requirements definition language that can be implemented in popular Modeling and Simulation tools (such as, Modelica) hiding the complexity of temporal logic behind an effective (visual, as in the developed libraries) representation of main requirements modeling constructs (see Sects. 4 and 5).

The aim of the proposal is to provide an integrated approach and toolchain for representing requirements in terms of a requirements model that can be bound with architectural and behavioral models in a common modeling and simulation environment (such as, Modelica) so as to evaluate requirements against the system design through simulation (see Sect. 6)

4 Verification of system requirements through simulation: an integrated solution

This section presents a new integrated solution to automate the verification of system requirements through simulation. Figure 2 delineates its main parts.

The solution is essentially based on three different kinds of models and related bindings: (i) "Requirements model", which provides a formal representation of the system requirements; (ii) "Architectural models", which represent the system structure at different decomposition levels (e.g., system, subsystems, equipment, components); (iii) "behavioral models", which specify the behavior of system components and their interactions. These models can be linked together by using different binding techniques so as to perform several analyses throughout the systems engineering process. As an example, it is possible to evaluate different system design alternatives against the requirements through simulation and potential (emerging) behaviors that can lead to requirement violations under given state conditions (e.g., resources unavailability, errors, and failures at both system and component level).

Through the formal definition of requirements and the bindings between the parts, the solution supports the *"Forward from the requirements"* tracing mode (see Subsection 3.3) by ensuring that each requirement is linked to a system component. However, a *requirements trace matrix* is needed to represent in a non-ambiguous way the links between requirements and system components.

With reference to the system life cycle stages identified in the IEEE Std. 24748-1-2011 (see Fig. 1) [62], the proposed solution offers benefits in the following stages:

- *Concept stage —System Requirements Definition*. To establish automatic relationships across requirements to evaluate the validity and ensure coherence between them;
- *Development stage —Design Definition*. To support the decision-making process in determining the best design by evaluating different design alternatives against requirements;
- *Development stage—Verification and Validation*. To define a digital twin to improve and monitor the entire CPS. The digital twin technology facilitates visibility in the system operations and allows to perform what-if analysis. This is performed through simulation by considering different conditions that may be otherwise impracticable to recreate by using the real CPS;
- *Utilization and Support stages—Operation and Maintenance*. To verify whether changes introduced to the CPS in operation (such as patches that may degrade performance and affect functional requirements) lead to a system that still meets the requirements.

Concerning the simulation process, three types of data can be used for feeding the requirement models: (i) *Historical data*, past performance data of the overall system and individual components; (ii) *Real data*, data coming from the system in operation, i.e., from sensors and actuators, outputs of components disseminated across the production chain including outputs of supplementary business systems; (iii) *Synthetic data*, data from engineers, machine learning and artificial intelligence systems.

A real-world CPS comes in the form of an ecosystem of components that are necessary to represent and study its behavior by considering the myriad of operating conditions. For each of them, the entire CPS must always meet the requirements and continue its operation. Through the help of the proposed solution, it is possible to conduct these analyses.

The following subsections present the proposed solution along with the details on how it is possible to handle a requirements model. Specifically, Subsection 4.1 presents the transformation process that allows deriving formal requirements starting from their description in natural language. Subsection 4.2 describes how to simulate the so-obtained formal requirements. Finally, the requirement verification procedure is presented in Subsection 4.3.

4.1 From natural language to formal requirements

In CPS, requirements typically concern the dynamic behavior of a system. They are expressed as a set of constraints bearing on objects and involving physical time and events. Expressing requirements as constraints leaves space for innovation and optimization. Innovation is the ability to find new solutions to a problem whose limits are expressed as a set of spatiotemporal constraints. Optimization is the ability to find the best solution among the possible newly identified solutions.

Requirements can be of several types. Three of them are of particular importance regarding CPS: safety, environmental, and economic, as the challenge is to build and operate safe, efficient, and environmental friendly systems. Many disciplines can be involved, one of the most important being physics, together with human factors, control engineering, stochastic aspects, etc.

Also, in the design and operation of CPS, many stakeholders are involved. They may enter and leave the project over long periods of time. Among them, the safety authorities are of particular importance because they have veto power on the engineering and operation decisions, and they cannot be bypassed (e.g., by turning to another competing safety authority with less stringent requirements). Therefore,

the ability to demonstrate the correctness of the design (i.e., its compliance with respect to all requirements) is as important as the design itself. This aspect is often overlooked, mainly because of the lack of elicitation methods and tools.

These aspects are the main drivers for FORM-L, as a language and a method for the formal capture and simulation of requirements, all along the engineering lifecycle, in particular covering the very early stages and the detailed design phases. The FORM-L language is described in the next section.

Requirements are always initially expressed using natural language. Therefore, simulating requirements results from a complex process that transforms natural language expressions into a formal executable model that can be used to automatically detect design errors.

The transformation process is similar to the design process. It starts from assumptions that describe the invariants of the system (i.e., its environment) and consists mainly of refinement and elicitation: one usually starts from very broad and general statements that are progressively refined until a first sketch of the system architecture is obtained, which is in turn decomposed into subsystems and components of different engineering domains (Hydraulics, Electro-mechanics, Telecom, and Instrumentation and Control - I&C). Refinement narrows the solution space by issuing new requirements. The challenge is to make sure that the new requirements are compatible with the existing ones, and that narrowing the solution space does not eliminate good potential solutions. The two challenges can be dealt with by automating the verifications and by exploring alternative solutions, using a simulator that captures all facets of the system of interest to the designer. Because the design process depends on the initial assumptions, it is often necessary to explore different scenarios on the possible assumptions when the engineering process spans over a very long time period. This amounts to repeating the design process for each scenario.

Therefore, the process of producing the simulator is very similar to the process of producing the real system. This is why the simulator can be thought of as a digital twin of the system.

This transformation process cannot be entirely automated because natural languages are rich and ambiguous. Richness implies that all aspects of an informal requirement cannot be captured in a formal requirement, or even in a set of formal requirements. Ambiguity means that there is no one-to-one relationship between informal and formal concepts. Also, as seen above, producing the simulator is very similar to designing the real system. It follows that if one could completely automate the generation of the simulator, then one could completely automate the design of the system itself.

As automating the verifications is an important aspect to ensure at each step that the design is compliant with the requirements, it turns out that the partial simulators constructed at each step are necessary to complete the full engineering cycle rigorously and eventually construct the full simulator for the final system. Therefore, if the design cannot be automated, how is it possible to automate the verifications?

First, one can notice that the verification problem is in general much simpler than the design problem. This is recognized in computational complexity theory that states that the verification of a solution is much faster than the search of a solution, as it is the case of NP-complete problems that (by definition) can be verified in polynomial time, but cannot be solved in polynomial time (if the conjecture $NP \neq P$ is indeed true).

The idea for automating the verification is to start from a formal description of the requirements that are as close as possible to their expression in natural language. FORM-L has been designed with this goal in mind. Going from sheer natural language expressions to formal expressions close to natural language can be achieved by the use of ontologies. This aspect will not be developed here.

FORM-L requirements express constraints on objects that depend on time, and that must be fulfilled during given time periods. The possible solutions must be found within the spatiotemporal (4D) space limited by the constraints. A given requirement is expressed in the form of a sentence that is constructed by assembling snippets of four different types:

- *WHERE* (spatial locator). This defines which objects in the system are subject to the requirement. The objects are defined by the architecture of the system that can be refined when moving from one design step to the next;
- *WHEN* (time locator). It defines the time periods when the requirement should be fulfilled. The time periods' boundaries correspond to events occurrences; a requirement without a time locator means that it should be fulfilled during the entire execution;
- *WHAT* (condition). It defines the condition to be fulfilled under *WHERE* and *WHEN*. It is a Boolean expression that defines a constraint on the properties of the objects involved in the *WHERE*;
- *HOW_WELL* (probabilistic constraint). It defines the probability that the *WHAT* must be verified (be true). *HOW_WELL* is made necessary due to the fact that no condition can be always verified under any circumstances: any system of components exhibits some probability of failure that must be taken into account in order to specify realistic systems that can be built according to realistic economic constraints (following the rule of thumb that the smaller the probability of failure, the higher the cost of the system).

Three examples of FORM-L statements are reported in List. 4, List. 5, and List. 6. Concerning examples in List. 4: the `during` keyword is used to indicate the time intervals when the conditions specified in the WHAT part must be verified; the `check` keyword states that the condition specified in the WHAT part must be verified, at the latest, at the end of the considered time interval. Concerning the example in List. 5 and 6: the `while` keyword indicates that the time period for the evaluation of the requirement lasts until a given condition holds true; the `ensure` keyword states that the condition specified in the WHAT part must be verified all along the considered time period. Further examples of requirements expressed in FORM-L are reported in Sect. 6 where also the `after` keyword is used to specify that a requirement must be evaluated after a condition becomes and holds `true` for a given time interval. Additional examples and a complete description of the FORM-L syntax and semantics can be found in [15, 18].

```
//During heating phases the maximum recirculated airflow
    of the aircraft ECS shall not exceed x% (max ratio) of
     the hot airflow entering the cabin
requirement maximumECSAirflow =
 during (operational and ECS.state==heatingPhase) //WHEN
 check (Ratio_max=<Ratio_max_req);                //WHAT
```

Listing 4 The maximum Environmental Control System (ECS) Airflow requirement

```
//When BPS is under maintenance, it must be not active
requirement bpsMaintenace =
 while (operational and BPS.state==maintenance) //WHEN
 ensure not (BPS.state==active);                //WHAT
```

Listing 5 The Backup Power System (BPS) maintenance requirement

```
requirement nominalOutputVoltageRange =
  while BPS.state==active              //WHEN
  ensure VOut in [minVOut, maxVOut];   //WHAT
```

Listing 6 BPS output voltage requirement

4.2 Simulating FORM-L requirements using ETL

Given a FORM-L requirement of the form:

$$R = [WHERE][WHEN][WHAT][HOW\_WELL] \quad (1)$$

What is the value of R? This question must be answered in order to simulate a system, and evaluate the satisfaction of the FORM-L requirements during the simulation. Given two requirements R_1 and R_2, how can they be combined to produce a third requirement $R_3 = f(R_1, R_2)$? Such a question must be answered to simulate complex requirements built from simpler ones.

An answer to those questions is given by the *Extended Temporal Language (ETL)* which is aimed at simulating the temporal aspects of FORM-L, i.e., the *WHEN* and the *WHAT*, given the *WHERE* and the *HOW_WELL*, and using 4-value logic [63]. It is worth noting that ETL is not meant to be used directly by the end-user, but as a means to simulate models expressed in FORM-L that as a high-level language for requirements modeling, has been especially conceived for practitioners. Indeed, ETL enables to express real-time constraints on continuous physical variables and state events, and consequently handle several real-time threads. Thus, the idea is to automatically generate ETL expressions for the evaluation of the temporal constraints expressed in FORM-L models [63]. When translating FORM-L expressions into ETL constructs (see [63] for the ETL complete syntax and semantics), only the *WHEN* and *WHAT* parts are effectively translated. They are made of *ETL* expressions that use *external* variables bound to the executable model of the system architecture using so-called *bindings* (see Fig. 3).

The model of the system architecture is the *behavioral model* of the system equipped with virtual sensors called *observers*. The role of the observers is to translate physical notions (such as *flows* or *potentials*) into functional ones (such as *in operation* or *switched on*). It appears here that

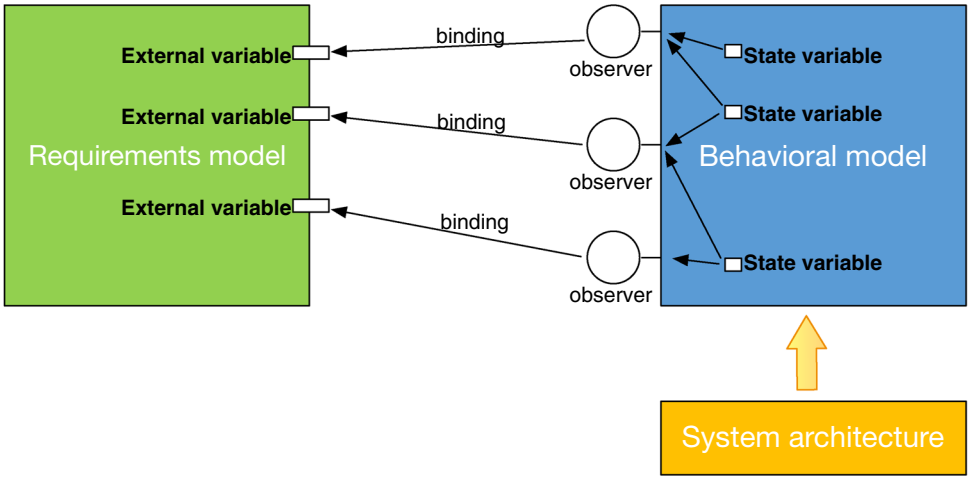

Fig. 3 Bindings

some kind of behavioral model is always needed. It can be of any degree of complexity, ranging from simple intervals (min and max values) to finite state machines (such as the SysML state diagram), detailed 1D physical models (such as Modelica models), or even 3D physical models (such as CFD models). The behavioral model can be deterministic (such as Modelica models expressed with hybrid differential equations) or non-deterministic (such as Stimulus[1] models expressed with stochastic timed automata).

In ETL, the value of a requirement is true if it is satisfied and false if it is not satisfied (or synonymously, violated). However, one must take into account the fact that requirements are subject to time periods. At time instants outside of a time period, a requirement is undefined, which means that it is not applicable. At time instants inside a time period, it is not always possible to tell whether a requirement is satisfied because one must wait inside the time period for the occurrence of a particular event to make this decision. This particular event is called the *decision event*. Often, the decision event is the end of the time period itself, but it can be a threshold crossing or any other kind of event. Before the decision event, the requirement is undecided. Therefore, requirements take their values in the set $\mathbb{P}_4 = \{undefined, undecided, false, true\}$.

By definition, a requirement R is denoted by Eq. 2.

$$R \overset{\text{def}}{=} \varphi \otimes P \tag{2}$$

φ is the denotation of the *WHAT*, similarly P is the denotation of the *WHEN*. The sign $\otimes$ expresses the fact that R is obtained by composing φ with P.

[1] https://www.argosim.com/home/stimulus-for-requirements/.

The value of R is denoted as $val(R)$, which takes its values in $\mathbb{P}_4$. As a consequence, R is satisfied when $val(R) = true$. $val(R)$ depends on the history of φ. $\varphi \otimes P = (\varphi \otimes P)(t)$ is an expression of time that is defined over time period P, where $t \in \mathbb{N}$ represents a time instant. Outside of P, it takes by definition the value defined by Eq. 3:

$$t \notin P \Leftrightarrow (\varphi \otimes P)(t) = undefined \tag{3}$$

Then, computing the value of R is similar to integrating $R = (\varphi \otimes P)(t)$ over the duration of the simulation (4).

$$val(R) = \int d((\varphi \otimes P)(t)) \tag{4}$$

$d((\varphi \otimes P)(t))$ acts like a differential operator in continuous time that extracts events from φ by comparing to consecutive values of φ along the time (the details can be found in [63]).

For a discrete clock, the continuous sum $\int$ must be replaced by the discrete sum $\sum$. The sum is evaluated according to a truth table on $\mathbb{P}_4$ that implements the requirements evaluation rules. The truth table for the sum operator is reported in Table 1. Similarly, the truth tables for the other operators (e.g., $\wedge$ and $\neg$) can be found in [63].

These rules state, for instance, that when a requirement is violated, it cannot subsequently be satisfied, or when

Table 1 The truth table for the sum operator $\varphi_1 + \varphi_2$

φ_1				
φ_2	True	False	Undecided	Undefined
True	True	False	True	True
False	False	False	False	False
Undecided	True	False	Undecided	Undecided
Undefined	True	False	Undecided	Undefined

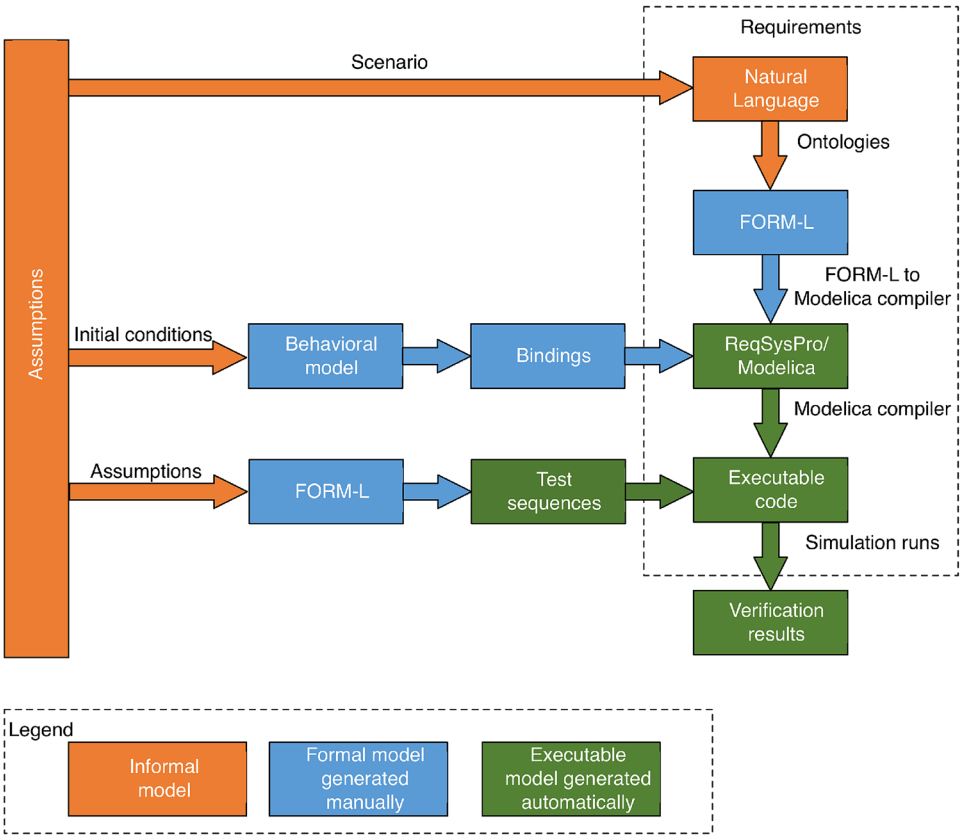

Fig. 4 Requirement verification process

a requirement is undecided, it remains undecided until it becomes true or false following the decision event. The value of $(\varphi \otimes P)(t)$ at each time instant t depends on φ and P. For instance, if φ checks whether a number of events must be greater than a given integer n inside P (event counting starts when P begins and ends when P finishes), then $(\varphi \otimes P)(t)$ stays undecided inside P until either the number of events becomes greater than n (then it becomes *true*), or until the end of P (then it becomes *false*).

If P has no duration, then $R = val(R)$. Therefore, R evaluates to itself if the decision whether R is satisfied can be made instantaneously. Otherwise, in general $R \neq val(R)$ which means that the evaluation of R is delayed with respect to the current value of R (i.e., at time instant t, R can be *true* or *false* and $val(R)$ still *undecided* as the evaluation time interval P is still not ended).

In order to combine requirements, $\mathbb{P}_4$ is equipped with the standard Boolean operators that follow the Morgan laws $a \vee b = \neg(\neg a \wedge \neg b)$, $a \wedge b = \neg(\neg a \vee \neg b)$ and $\neg\neg a = a$. All classical Boolean operators can be defined, e.g., $a \Rightarrow b \stackrel{\text{def}}{=} \neg a \vee b$. The truth tables for these operators can be found in [63]. Applying Boolean operators on elements of $\mathbb{P}_4$ has the same meaning as applying Boolean operators on elements of $\mathbb{P}_2 = \{true, false\}$, with the difference that at some time instants the result of the operation may be undefined (if outside of time periods and never evaluated inside a time period), or undecided (if inside a time period but still awaiting a decision event). The other difference is that the tautology $a \wedge \neg a = false$ is not always verified (because a

Table 2 Monitoring result of requirement block

Input is	Requirement is
Satisfied at least once and is never *Violated*	*Satisfied*
Violated at least once	*Violated*
Otherwise	*Untested*

may be *undefined* or *undecided*). The value of requirement $R = R_1\ op\ R_2$, where *op* is a Boolean operator, is obtained using the rule $val(R_1\ op\ R_2) \stackrel{\text{def}}{=} val(R_1)\ op\ val(R_2)$.

If φ is provided by an observer looking at the physical system, then φ will usually take the values *true* or *false*, e.g., feed pump in operation or not. However, it may sometimes take the value *undecided* if the observer cannot decide on the actual value, e.g., the reactor current operating point is too close to the boundary of the authorized operating domain to decide whether it is inside or outside the operating domain given measurements uncertainties. Finally, φ takes the value *undefined* outside of a time period P.

φ can also represent the value of a requirement; this means that $R = \varphi \otimes P(t)$ is a requirement on the value of a requirement; it is so possible to express generic requirements such as "As long as requirement $A.1$ on system A is satisfied, n seconds after requirement $B.1$ of the system B becomes violated, then requirement $C.1$ on system C should be satisfied". Requirements $A.1$, $B.1$ and $C.1$ can be inserted into the generic requirement depending on the detailed design of the system, and without modifying the generic requirement. This can be used, for instance, to express mission changes. It is also possible to express formal refinement schemes (or patterns, or procedures, or rules) such as $A_1 \Rightarrow R_1 \Rightarrow R_2$ which means that requirement R_2 should comply with requirement R_1 (i.e., the satisfaction of R_2 does not contradict the satisfaction of R_1), which should comply with assumption A_1 (A_1 is an assumption because it does not depend on any other requirement). Such patterns, or even more complex ones, can be used to express requirements on the design methodology. It is also possible to express generic statements of the contract theory from [64], such as the parallel composition of contracts $\mathfrak{C}(A, G) = \mathfrak{C}_1(A_1, G_1) \otimes \mathfrak{C}_2(A_2, G_2) : G = G_1 \wedge G_2, A = (A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2)$, where A_i denotes the assumptions and G_i denote the guarantees. This can be used to help stakeholders coordinate with each other and reach a common agreement.

4.3 Automating verifications: still a blend of manual and automatic actions

The intent of automating the verifications is the ability to automatically propagate the impact of modifications in the assumptions or in the requirements downstream to the design process. Modifications in the assumptions can be motivated by exploring alternative scenarios. Modifications in the requirements can be motivated by detecting design violations of the requirements when inspecting the verification results.

Figure 4 shows a requirement verification process that is a mixture of manual and automatic actions. Three kinds of models are involved: informal models, formal models manually derived from informal models, and executable models generated automatically from formal models. The idea is to generate automatically from FORM-L models the executable model that can observe automatically the behavior of the system under design and detect possible requirement violations. The FORM-L model is generated manually from the requirements expressed in natural language (the orange blocks are outside the scope of this paper). From the FORM-L model, test sequences could be generated automatically (this feature is currently not available).

The executable model that simulates the requirements is also generated from the same FORM-L model (this feature is already available in a form of prototype compiler developed by Inria and Sciworks Technologies that translates FORM-L models into Modelica models, cf. List. 7 and List. 8). It is a Modelica model constructed by assembling components from the ReqSysPro Modelica library developed by EDF R&D (cf. Subsection 5.2).

```
// A simple pumping system.
property model Pumping_system_set_requirement
 constant Real dur is 10.0;

 class Pump
  external Real pressure;
  fixed Real threshold_low;
  fixed Real threshold_high;
  external Boolean operator_start_pump;
 end Pump;

 external Real p1_pressure end p1_pressure;
 external Boolean p1_operator_start_pump end
     p1_operator_start_pump;

 Pump pump1 is new Pump (
   pressure is p1_pressure;
   threshold_low is 10.0;
   threshold_high is 100.0;
   operator_start_pump is p1_operator_start_pump;
 );

 external Real p2_pressure end p2_pressure;
 external Boolean p2_operator_start_pump end
     p2_operator_start_pump;
 Pump pump2 is new Pump (
   pressure is p2_pressure;
   threshold_low is 20.0;
   threshold_high is 100.0;
   operator_start_pump is p2_operator_start_pump;
 );

 external Real p3_pressure end p3_pressure;
 external Boolean p3_operator_start_pump end
     p3_operator_start_pump;

 Pump pump3 is new Pump (
   pressure is p3_pressure;
   threshold_low is 30.0;
   threshold_high is 100.0;
   operator_start_pump is p3_operator_start_pump;
 );

Pump {} all_pumps is { pump1, pump2, pump3 };

 requirement no_over_pressure is
   //WHERE
  for all p in all_pumps
   //WHEN
   after p.operator_start_pump becomes true for dur
   //WHAT
   ensure p.pressure < p.threshold_high;

end Pumping_system_set_requirement;
```

Listing 7 FORM-L model stating "The pressure of all pumps must not exceed a threshold after they have been started and for a given time duration"

```
//Formlc compiled file from source file "
    pumping_system_set_requirement2.fml"
model Pumping_system_set_requirement

 constant Real dur = 10.0;

 class Pump
  ReqSysPro.Interfaces.BooleanInput operator_start_pump;
  ReqSysPro.Interfaces.RealInput pressure;
  parameter Real threshold_high;
  parameter Real threshold_low;
 end Pump;

 ReqSysPro.Interfaces.RealInput p1_pressure;
 ReqSysPro.Interfaces.BooleanInput p1_operator_start_pump
     ;

 Pump pump1(
  operator_start_pump = p1_operator_start_pump,
   pressure = p1_pressure,
   threshold_high = 100.0,
   threshold_low = 10.0
 );

 ReqSysPro.Interfaces.RealInput p2_pressure;
 ReqSysPro.Interfaces.BooleanInput p2_operator_start_pump
     ;

 Pump pump2(
  operator_start_pump = p2_operator_start_pump,
  pressure = p2_pressure,
  threshold_high = 100.0,
  threshold_low = 20.0
 );

 ReqSysPro.Interfaces.RealInput p3_pressure;
 ReqSysPro.Interfaces.BooleanInput p3_operator_start_pump
     ;

 Pump pump3(
  operator_start_pump = p3_operator_start_pump,
  pressure = p3_pressure,
  threshold_high = 100.0,
  threshold_low = 30.0
 );

 Pump [3] all_pumps = {pump1, pump2, pump3};

 model Model__no_over_pressure_single
  ReqSysPro.Conditions.Ensure no_over_pressure__check;
  ReqSysPro.Interfaces.BooleanInput
      no_over_pressure__check__u;
  ReqSysPro.TimeLocators.Continuous.AfterFor
      no_over_pressure__ctl;
  ReqSysPro.Interfaces.BooleanInput
      no_over_pressure__ctl__u;
  ReqSysPro.Interfaces.RealInput
      no_over_pressure__ctl__duration;
   equation
    no_over_pressure__ctl.u = no_over_pressure__ctl__u;
    no_over_pressure__ctl.duration =
        no_over_pressure__ctl__duration;
    no_over_pressure__check.u =
        no_over_pressure__check__u;
    connect (no_over_pressure__ctl.u,
        no_over_pressure__check.u);
 end Model__no_over_pressure_single;

 parameter Integer nz_all_pumps = size (all_pumps, 1);
   Model__no_over_pressure_single[nz_all_pumps]
      no_over_pressure_p;
   ReqSysPro.Conditions.Operators.And4_n no_over_pressure
      (N = nz_all_pumps);
   equation
    for i in 1:nz_all_pumps loop
     no_over_pressure_p[i].no_over_pressure__check__u =
         all_pumps[i].pressure < all_pumps[i].
         threshold_high;
     no_over_pressure_p[i].no_over_pressure__ctl__u =
         all_pumps[i].operator_start_pump;
     no_over_pressure_p[i].no_over_pressure__ctl__duration
          = dur;
    end for;

    for i in 1:nz_all_pumps loop
     no_over_pressure.u[i] = no_over_pressure_p[i].
         no_over_pressure__check.y;
    end for;

 end Pumping_system_set_requirement;
```

Listing 8 Automatic translation in Modelica/ReqSysPro of the FORM-L model in Listing 7

The so-obtained executable model is then semi-automatically bound to the behavioral model using the binding process described in [65]. The executable code for the full simulator is generated using a Modelica compiler. The simulation runs take as inputs the automatically generated test sequences and the initial conditions calculated from the assumptions.

The results are provided in the form of a list of requirements that are satisfied (*true*), violated (*false*), not tested (*undefined*, as nothing can be said about the requirement satisfiability) or incompletely tested (*undecided*, as the requirement could have been satisfied/violated during the test but, as the test has not been completed, it is not possible to say if, at the end of the test, the requirement would be satisfied/violated). This principle has been successfully used at EDF to (partially) automate the FMEA of a HVAC system (see Subsection 6.2).

5 Libraries for the simulation of requirements

This section presents two libraries for handling and simulating FORM-L requirements: (i) The *Modelica_Requirements* library; (ii) The Modelica *ReqSysPro* Library.

The Modelica_Requirements library (Section 5.1) is an open-source Modelica library, developed by the partners involved in the MODRIO project [10], that implements a subset of the FORM-L language in the form of Modelica blocks.

The Modelica ReqSysPro library (Section 5.2), developed by EDF R&D, exploits the ETL-based approach (see Section 4.2) to simulate a system, and evaluate the satisfaction of FORM-L requirements during the simulation.

5.1 The Modelica_Requirements library

The Modelica_Requirements library[2] allows for the definition of the FORM-L requirements by drag-and-drop operations of graphical blocks. It is organized in 15 sub-packages (see Fig. 5), each of which provides functionalities to allow the formal definition of requirements and their verification during the simulation. In total, there are about 60 input/output blocks to define requirements graphically. A brief description of the most important sub-packages is given below. A detailed description of the library can be found in [19].

– *Verify*. Blocks to define requirements and evaluate whether they are *satisfied*, *violated*, or *not tested* during simulation. It also offers the *PrintViolation* block that

[2] https://github.com/modelica-3rdparty/Modelica_Requirements.

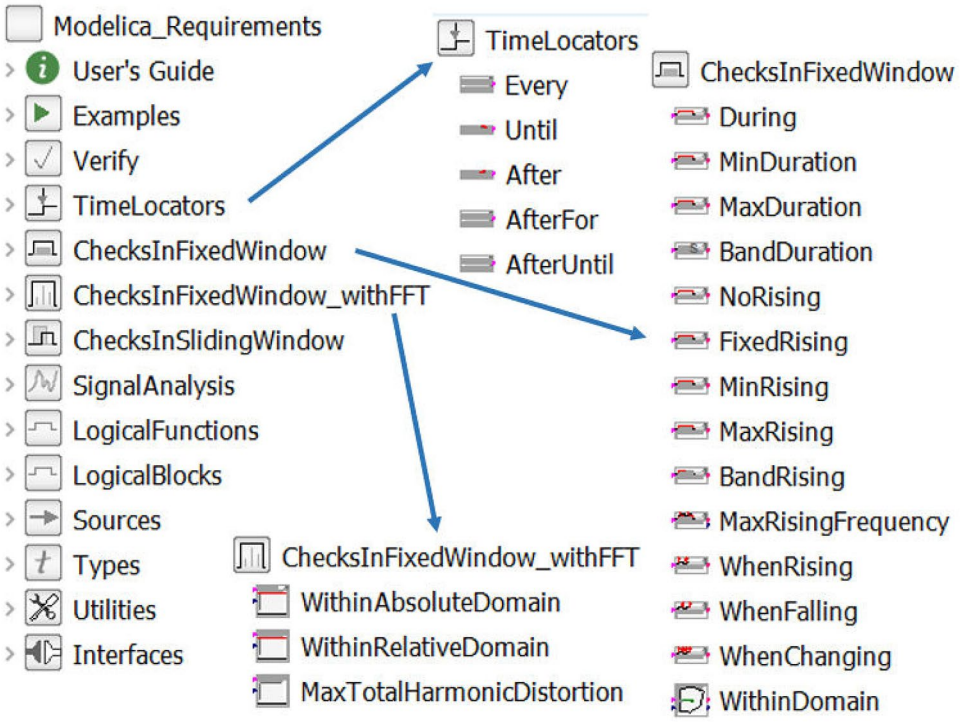

Fig. 5 The Modelica_Requirements library

allows storing, into a log file, the information concerning the configuration parameters and status of all requirements;
- *Time Locators*. Temporal operator blocks for defining time intervals of interest (see Fig. 5);
- *ChecksInFixedWindow*. Blocks that are useful to determine whether a particular property is fulfilled or not in a given time window (see Fig. 5);
- *ChecksInFixedWindow_withFFT*. Blocks that check properties based on FFT (Fast Fourier Transform) computations in fixed time windows. An FFT determines the frequency content and amplitudes of a sampled, periodic signal, and the blocks in this package check whether these frequencies and amplitudes fulfill certain conditions (see Fig. 5);
- *ChecksInSlidingWindow*. Blocks that allow determining whether a property is fulfilled or not in a given sliding window (e.g., if a sliding window has size T and t represents the current time instant, then in every time range $[t - T, t]$ the property must be fulfilled).

The library supports two-valued logic with the standard Modelica type *Boolean* and a restricted form of three-valued logic with the user defined type *Property* that can have values *Satisfied* (true), *Violated* (false) or *Undecided* (undefined or undecided). Typical blocks check whether for a fixed or sliding window defined by various conditions (= *WHEN*) a provided *Boolean* input is *true* (= *WHAT*). When the conditions are *true*, the block outputs *Satisfied* if the input is *true* and outputs *Violated* if the input is *false*. If the conditions are *false*, the block outputs *Undecided*. The *Property* outputs of these blocks can be combined with logical conditions. Finally, there is a *Requirement* block that defines that the *Property* input is a required property.

This block, which is provided by the *Verify* package, monitors its input over a simulation and computes its status at the end of the simulation run according to Table 2.

The requirement *nominalTempRange* of the HVAC system (see Subsection 6.2) can be formulated with the Modelica_Requirements library as shown in Fig. 6.

The top-most *WithinBand* block *tempBand* has as input the room temperature *tempRi* and returns with its Boolean output signal *tempBand.y* whether this temperature is in the required temperature band or not. The middle *During* block *during* has as input the Boolean expression *plant.state* <> *accident and atmosphere.state* <> *heatwave* As long as this condition is true, it is monitored whether the *check* variable *tempBand.y* is *true*. The result is provided as *Property* output variable which is connected to the *Requirement* block *req_TemperatureRange*. The icon of this block displays the required property in textual form. At the end of

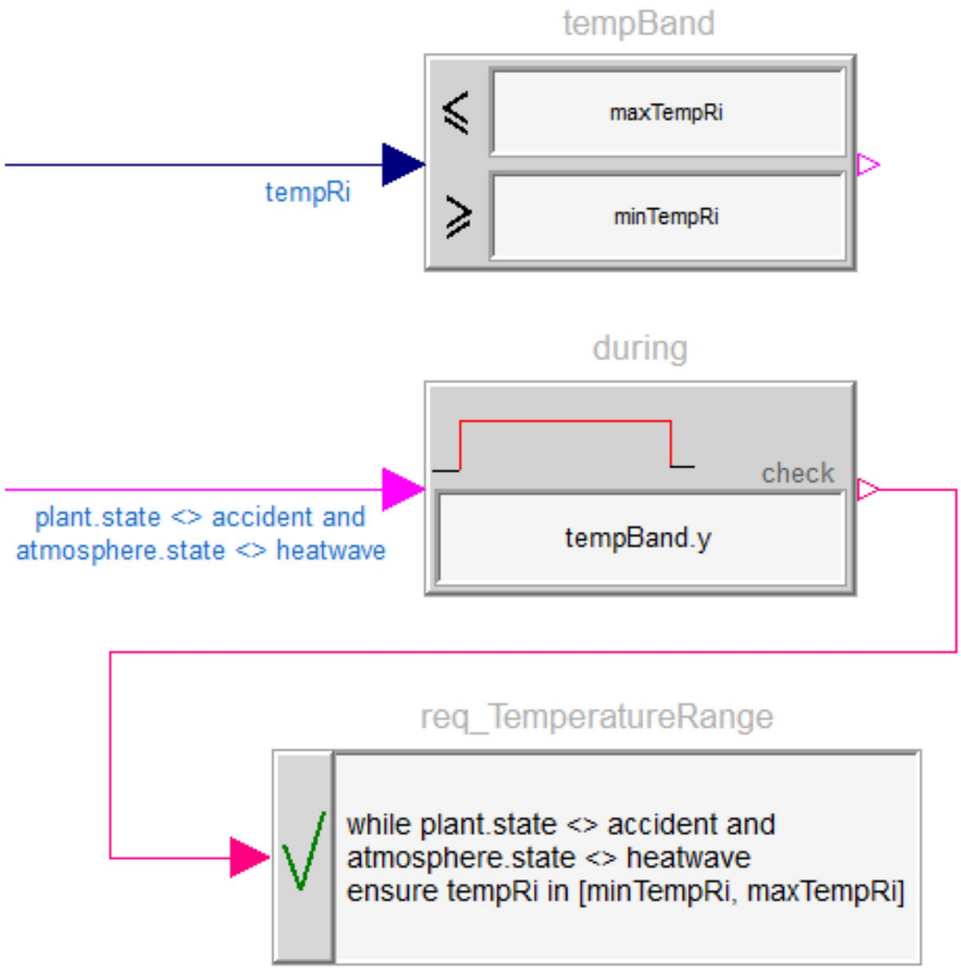

Fig. 6 FORM-L HVAC requirement for thermal conditioning for one room expressed with blocks of the Modelica_ Requirements library

a simulation, the status of all required properties is reported according to table 2.

Within the ITEA project MODRIO, the library was evaluated by Dassault Aviation with typical requirements for aircraft systems.

5.2 The Modelica ReqSysPro library

The Modelica ReqSysPro library allows evaluating FORM-L models expressed as ETL constructs (see Sect. 4.2). It offers a set of blocks representing basic ETL operators through Modelica constructs, ETL operators defined as a combination of lower-level ETL operators, and FORM-L operators expressed as ETL operators [63]. The library is composed of two main components *time locators* and *conditions* that can be combined to build the temporal parts of FORM-L expressions. Figure 7 presents the eight packages that compose ReqSysPro:

– *TimeLocators*. It provides the blocks for defining the *WHEN* with continuous or discrete clocks; as an example, the following *ContinuousTimeLocators (CTLs)* are available: *After*, which defines CTLs that begin with each occurrence of an event and last until the end of the simu-

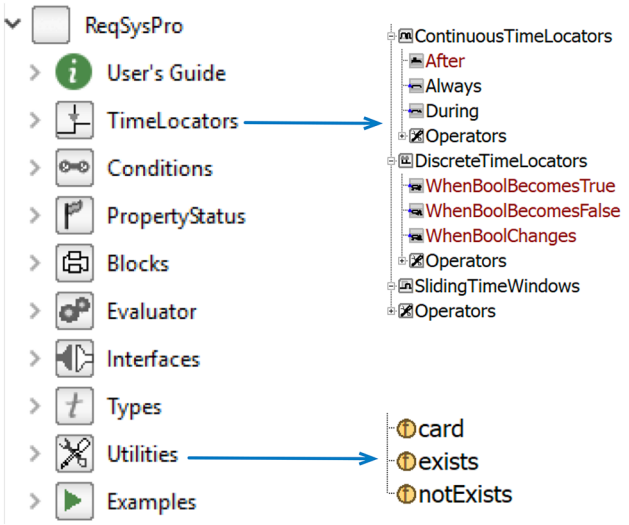

Fig. 7 The Modelica ReqSysPro library

lation run; *Always*, which defines a single CTL covering the complete simulation run; *During*, which define CTLs when the Boolean input is true;
- *Conditions*. It provides the blocks for defining the *WHAT*. It also provides the blocks for the 4-value logic (and, not, or, xor, implies, etc.);
- *PropertyStatus*. It provides the block for translating requirements values from {*undefined, undecided, false, true*} to {*NotTested, Violated, Satisfied*}.
- *Blocks*. It provides general-purpose blocks, such as: *BooleanPulse*, which generates a pulse signal of type Boolean; *IntegerConstant*, which generates a constant signal of type Integer; *EventPeriodicSample*, which generates periodic occurrences of an event.
- *Evaluator*. It implements Equation 4 for evaluating requirements.
- *Interfaces*. It defines the library connectors.
- *Types*. It defines the types used in the library. In particular it defines the requirements type as enumeration {*undefined, undecided, false, true*}.
- *Utilities*. It provides utility functions such as: *card*, which returns the number of elements of a Boolean vector that are true; *exists*, which returns true if at least one element of a Boolean vector is True; *notExists*, which returns true if no element of a Boolean vector is True.

The rationale for the Modelica *ReqSysPro* library along with the complete syntax and semantics of the implemented constructs for the verification of CPS requirements can be found in [63].

A requirement is built by connecting a time locator block to a condition block. Requirements can be connected together by using blocks for the 4-value logic. Figure 8 illustrates requirement "When the system is in operation, the pump should not be started more than twice, and when the pump is started, its temperature should never exceed 50°C". The inputs correspond to external variables. The output is the value of the requirement. 2-value logic variables are represented by purple connectors. 4-value logic variables are represented by brown, connectors.

Figure 9 shows a more abstract requirement R that states "Over a given time period, if requirement *R*1 is *not satisfied*, then requirement *R*2 should be *satisfied*". It can be used to express a mission change.

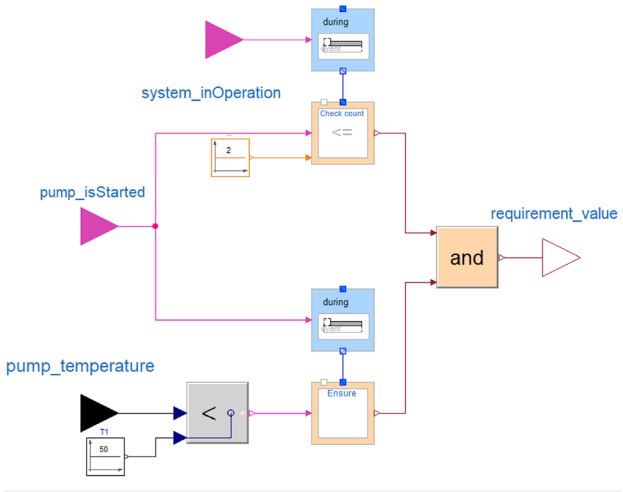

Fig. 8 Requirement expressed with ReqSysPro blocks

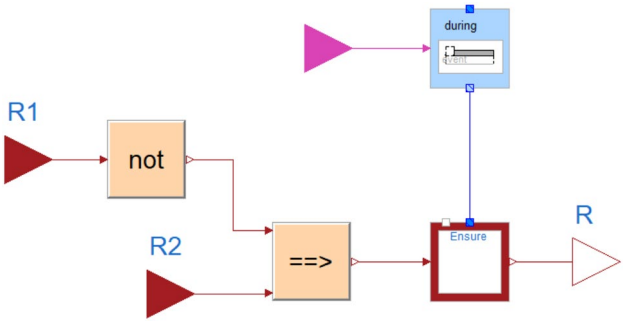

Fig. 9 Mission change expressed with ReqSysPro blocks

6 Reference scenarios

As presented in the previous sections, the proposed solution allows formally representing requirements and rules for verification of system constraints through simulation. To exemplify the introduced solution, two case studies concerning a Trailing-Edge High-Lift system of a commercial transport aircraft and the FMEA of an HVAC system are presented in the next sections. The first case study was chosen due to its complexity since it involved multiple partners involved in the MODRIO project with different viewpoints on how a Trailing-Edge High-Lift system should be developed. The proposed solution has been exploited to bind the system requirements with various behavioral models to evaluate different architectural designs through simulation by adopting the Functional Mockup Interface (FMI) standard [66, 67]. The second case study has been chosen due to significant differences from the first one, where the organizations' design perspective were important. It was chosen to show how the proposed approach can be exploited to handle the design of critical systems in combination with well-established techniques for reliability analysis (e.g., FMEA, FMECA).

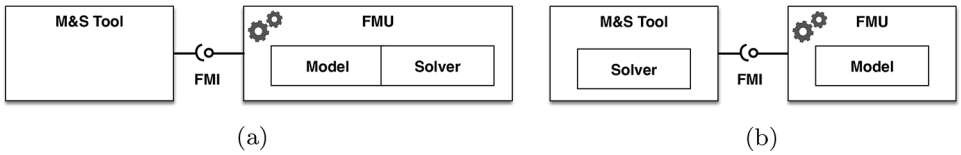

Fig. 10 The FMI standard defines an interface to enable both Co-Simulation (a) and Model Exchange (b) of an FMU

6.1 Case study: verification of a trailing-edge high-lift system

In the aircraft design process, a Trailing-Edge High-Lift system is a component or a mechanism mounted on an aircraft's wing that allows increasing the amount of lift produced by the wing when required according to Equation 5 [68].

$$L = \frac{1}{2} \cdot \rho \cdot V_{\infty}^2 \cdot S \cdot C_L \tag{5}$$

where:

L is the amount of produced lift;
ρ is the air density factor;
S is the lifting surface;
C_L is the lift coefficient;
V_{∞} is the velocity.

In this case study, the FMI was adopted to perform the Co-Simulation of the system components within the Siemens AMESim simulation environment [69]. FMI is a free standard that defines a container and an interface to exchange simulation models. It has been proposed by the Daimler AG within the ITEA2 project MODELISAR and the first version, namely FMI 1.0, was released in 2010 to improve the interoperability of simulation models among suppliers and Original Equipment Manufacturers (OEMs). The current stable version, which is FMI 2.0, extends the interface specifications to enable both Model Exchange (*FMI for Model Exchange - FMI ME 2.0*) and Co-Simulation (*FMI for Co-Simulation - FMI CS 2.0*) of simulation models. The main objective of the *FMI ME 2.0* interface (see Fig. 10a) is to allow any simulation environment to generate and export C/C++ code or binaries of a dynamic model as an input/output block to reuse it in other simulation environments. The behavior of the model is specified through differential, algebraic and discrete equations with time, states, and step-events, which are specified in the source code since the model does not have within it the solver. The aim of the *FMI CS 2.0* interface (see Fig. 10b) is to couple two or more dynamic models in a common co-simulation environment. This can be done because each model comes with its specific solver. This interface specifies functionalities for handling the communications between a Master Algorithm (MA) and a slave. Each slave has a pre-defined set of inputs and outputs that are known by the MA. MA is in charge of the configuration, coordination, and management of the slaves during their execution. The data exchange is limited to discrete communication points in time and the models are solved independently between these communication points. The standard does not specify any MA but its definition is in charge of developers.

Models adhering to the FMI standard are called *Functional Mock-up Units (FMUs)* [66, 70, 71]. More in detail, an FMU is a compressed file with extension *.fmu* that contains: (i) *Source code*, the C/C++ source code that specifies the behavior of the dynamic model, including runtime libraries; (ii) *FMI Model Description*, an XML file containing the specifications of all the exposed variables, static information used by the model during its execution, and the shared libraries to the target operating systems, such as Windows (*.dll), Linux (*.so) and Mac OSX (*.dylib); and, (iii) *Additional resources*, a folder containing further data, such as icons, supporting files, maps, and tables.

6.1.1 Requirements model

High-lift devices can be either movable surfaces or stationary components that are designed to increase lift during the take-off, initial climb, and landing phases of flight but they may also be exploited in any other low-speed situations. They allow decreasing the surface area of the wing, thus reducing its drag and making the aircraft more efficient in terms of fuel consumption, during the cruise phase of the flight.

Since the system under study has only aerodynamic purposes, its functional requirements are only related to its aerodynamic behavior. The top-level requirement of the high-lift subsystem is to rototranslate the flap surface during the different flight steps so as to obtain satisfactory performance of the aircraft (system level) [16]. Concerning the geometrical variables involved in the requirements, they are measured from a reference system on the flap surface.

Functional requirements define what a system is supposed to accomplish. The high-lift device shall position the flap surface to satisfy aerodynamic requirements during the phases of take-off and landing (2D kinematic characteristics *R1, R2, R3*).

– $R1_a$: The longitudinal position of the flap surface shall be within $[X_{min}, X_{max}]_{take-off}$ when the Deployment Angle (DA) is in $[D_{min}, D_{max}]_{take-off}$

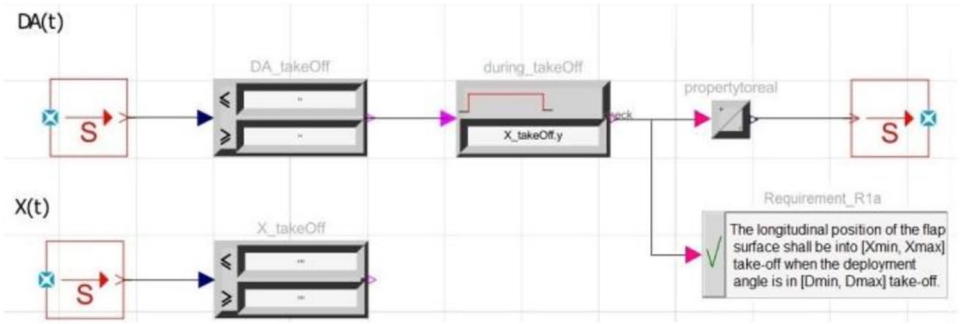

Fig. 11 Possible implementation of $R1_a$ with the Modelica_Requirements library

- $R1_b$: The longitudinal position of the flap surface shall be within $[X_{min}, X_{max}]_{landing}$ when DA is in $[D_{min}, D_{max}]_{landing}$
- $R2$: The lateral position of the flap surface shall be constrained during the longitudinal translation to avoid contact with the wing surface and to ensure a diagonal translation on the Y-X plane.
- $R3_a$: The vertical position of the flap surface shall be within $[Z_{min}, Z_{max}]_{take-off}$ when the longitudinal position is in $[X_{min}, X_{max}]_{take-off}$
- $R3_b$: The vertical position of the flap surface shall be within $[Z_{min}, Z_{max}]_{landing}$ when the longitudinal position is in $[X_{min}, X_{max}]_{landing}$

Performance requirements are introduced to evaluate and to compare the design variants regarding features of interest that should be considered during the design process (*R4, R5, R6*).

- *R4*: The sensitivity angle shall be constrained in order to ensure a smooth variation of the DA during the deployment. This variable is obtained as the first derivative of the DA with respect to the longitudinal position: $Y < K_{SA}$.
- *R5*: The height of the fairing shall be constrained so as to limit the aerodynamic drag. It represents the vertical distance between the bottom skin of the wing and the lowest point of the mechanism: $h(t) < K_{FH}$.
- *R6*: The maximum torque used by the motor shall be minimized to reduce the required motor size.

In the following, it is shown how the requirements $R1_a$ and *R5* have been defined in FORM-L. It is worth noting that $R1_a$ has been delineated through the FORM-L keywords *during* and *check*. The first one defines the time locators, which represent the time intervals when the conditions specified by *check* must be verified. In more detail, the *check* keyword states that the condition must be verified at the latest at the end of the time interval.

```
propertyModel Req_HL_device
 propertyModel R1
  external Real DA, X;
  parameter Real DA_min_Toff = 14, DA_max_Toff = 16;
  parameter Real X_min_Toff = 300, X_max_Toff = 400;
  required property R1_a =
   during (DA > DA_min_Toff and DA < DA_max_Toff)
   check  (X > X_min_Toff and X < X_max_Toff);
 end R1;

 propertyModel R5
  external Real h;
  parameter Real K_FH = 0.9;
  required property R_FH = check (h < K_FH);
 end R5;
end Req_HL_device;
```

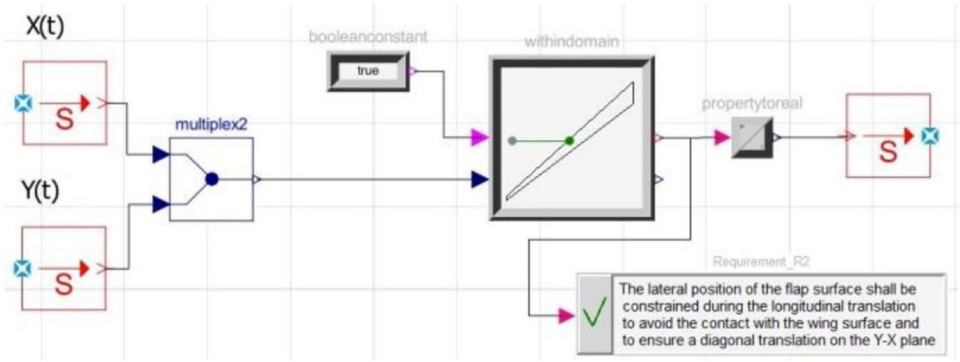

Fig. 12 Possible implementation of *R*2 with the Modelica_Requirements library

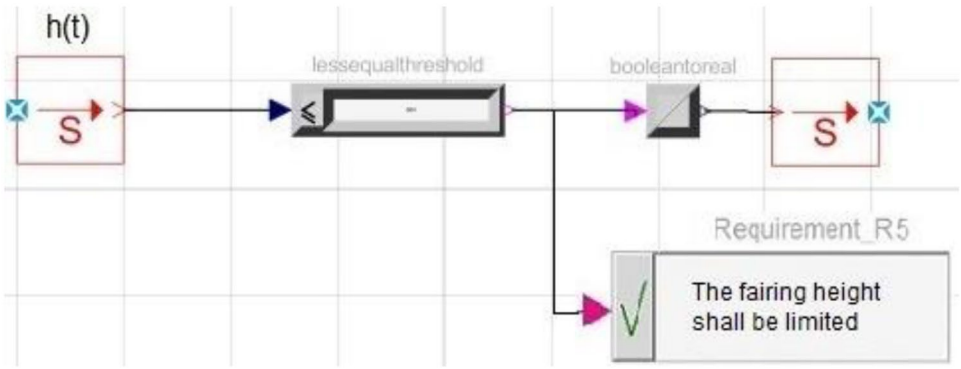

Fig. 13 Possible implementation of *R*5 with the Modelica_Requirements library

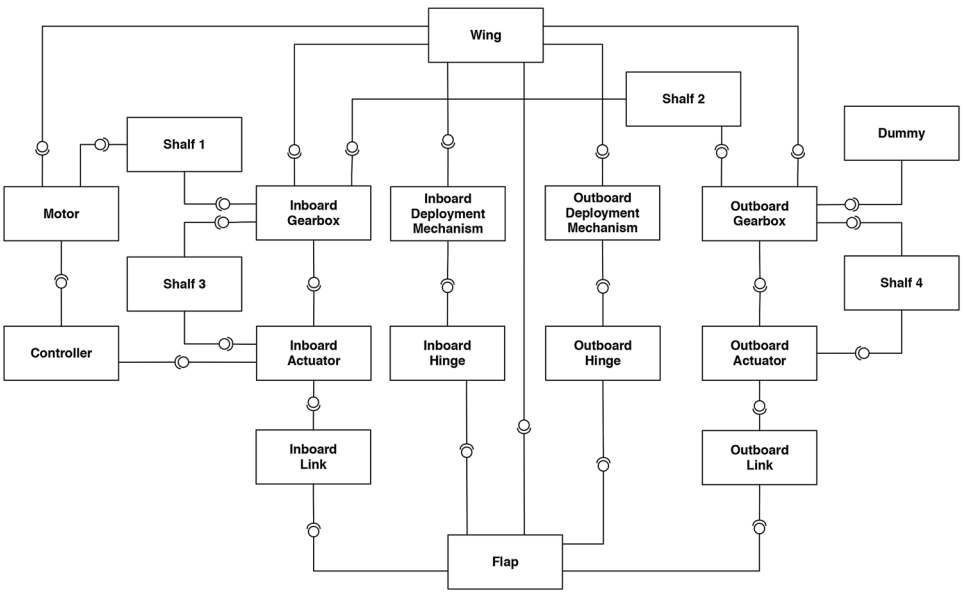

Fig. 14 Architectural model of the high-lift system

Figure 11 depicts the model of $R1_a$ where the *Real variables* are inputs of the *WithinBand* blocks. The *Boolean* output is true if the input is within the range specified by the parameters. Two signals (*condition* from *DA* and *check* from *X*) are generated and forwarded as inputs to the During block. *During* a *condition* phase the output value is *Satisfied* if *check* is *true*, *Violated* otherwise. When *condition* is *false* the output is *Undecided*, suggesting that the property is not tested.

Figure 12 shows the model of *R*2. The two *Real variables* are the longitudinal and lateral positions of the flap surface that are provided as inputs to the *WithinDomain* block. This latter block *checks* whether the 2D-input point (x, y) is within an area defined by a closed polygon. The shape of

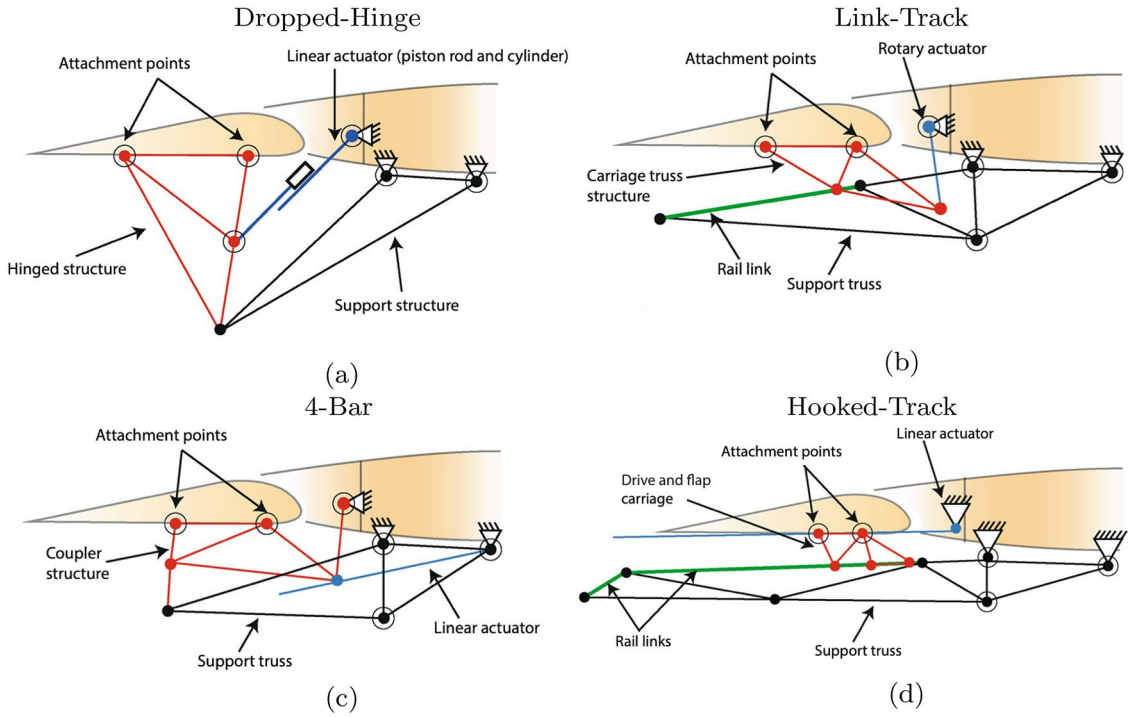

Fig. 15 CAD models of the deployment mechanisms that define the design variants of the system [72]

the polygon is defined by means of its vertices (see Fig. 12). The "Requirement_R2" block collects the status of *R*2 during the simulation.

Figure 13 depicts the *R*5 model. The *Real variable* represents the height of the fairing, which is given as input to the *LessThreshold* block. This latter block *checks* whether the input is less than a threshold specified as a parameter.

6.1.2 Architectural model

The architecture of the Trailing-Edge High-Lift system is composed of seven parts: a motor, a flap, deployment mechanisms, actuators, gearboxes, shafts, and a wing attachment structure. Four variants of the architecture have been defined with common but different mechanisms in implementing the deployment function. Figure 14 shows the system architecture.

Variants are realizations of the architecture with different component models or different characteristics to change design variables. Thus, there are four design variants of the system according to the kind of deployment mechanism (see Fig. 15) employed in the architecture [72]: (a) *Dropped-Hinge*, (b) *4-Bar*, (c) *Link-Track*, and (d) *Hooked-Track*; where, the blue line represents a *drive link*; the green line delineates a *rail link*; in red a *moving truss*; and finally, the black line is a *support truss*.

6.1.3 Behavioral model

The multi-body model of the high-lift system can be controlled by interacting with different parameters such as: (i) *kinematic* sizing; (ii) *geometry* for structural sizing; (iii) *finite element analysis* to consider flexibility of components [16] (see Fig. 15a).

6.1.4 Binding models

Once the system requirements have been modeled in FORM-L and implemented as Modelica sub-models, the binding with the system model can be made. The binding procedure has been made by exporting the involved parts as FMUs in co-simulation modality according to the FMI standard [66, 71]. The FMUs have been linked together and simulated by using the Siemens AMESim simulation environment (see Fig. 16). Many test cases can be performed, also allowing varying the test scenario and the system parameters.

6.1.5 Simulation and results analysis

The objective of the simulation is to evaluate the state of the requirements and compare the different design alternatives. In Fig. 17 a 2D representation of requirements $R1_a$ and $R1_b$ is shown along with the trajectory for the deployment of the four design variants.

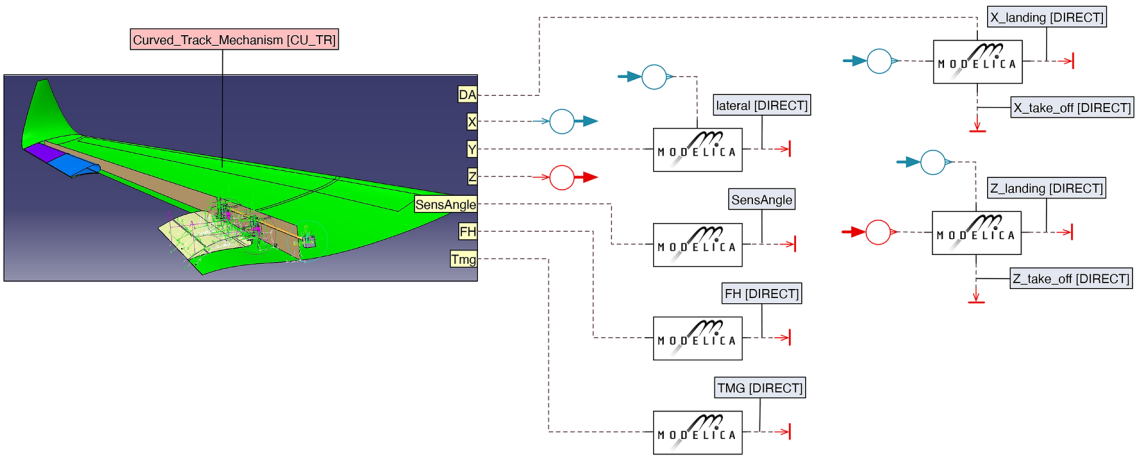

Fig. 16 Connection between system and property models in Co-simulation modality with a sub-system exported from the Modelica environment

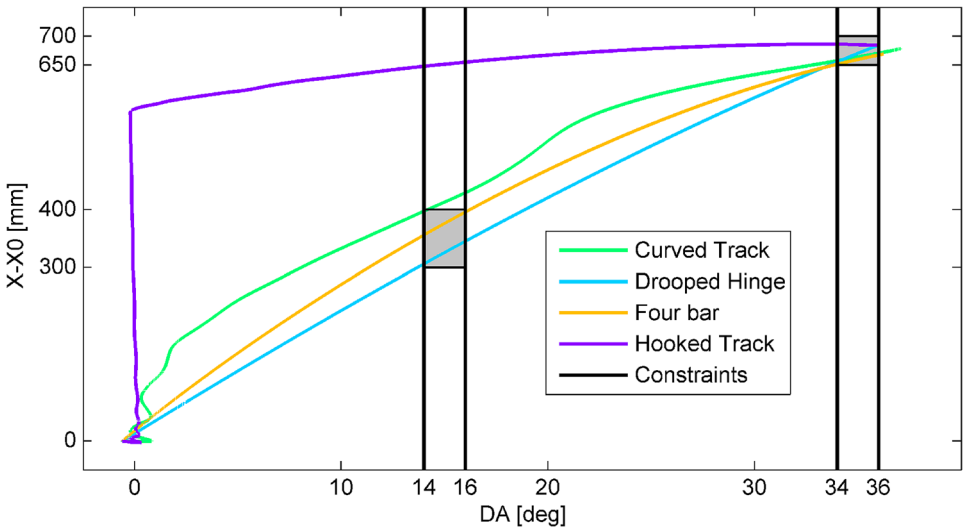

Fig. 17 Simulation of the four design variants with respect to the 2D representation of the requirements $R1_a$ (take-off scenario) and $R1_b$ (landing scenario)

Table 3 Design alternatives comparison

Property	System design			
	Dropped hinge	Four bars	Curved track	Hooked track
$R1_a$	✓	✓	✗	✗
$R1_b$	✓	✓	✓	✓
$R2$	✓	✓	✓	✗
$R3_a$	✓	✗	✓	✓
$R3_b$	✓	✓	✓	✗
$R4$	✓	✗	✗	✓
$R5$	✗	✗	✗	✓
$R6$	I	IV	II	III

The output of a simulation run for a system design shows the state of the requirements over time, for the input scenario, which is a combination of the system and requirements parameters.

Table 3 summarizes the results for the different design alternatives. Obviously, they depend on the chosen parameters and constraints. From the results, it emerges that the dropped hinge mechanism fulfills most of the requirements and requires the minimum motor torque to deploy the flap surface. The last row enumerates the design variants according to requirement $R6$, which is about the maximum torque employed by the motor (I stand for minimum torque, IV for maximum torque).

In Table 3, the following notation is used: if the system design fulfills the property, it is indicated with a check-mark ✓; otherwise, the ✗-symbol is used.

The library allows also generating textual reports of requirements assessment for each simulation run, which can be useful to build report documents when a large amount of tests is performed.

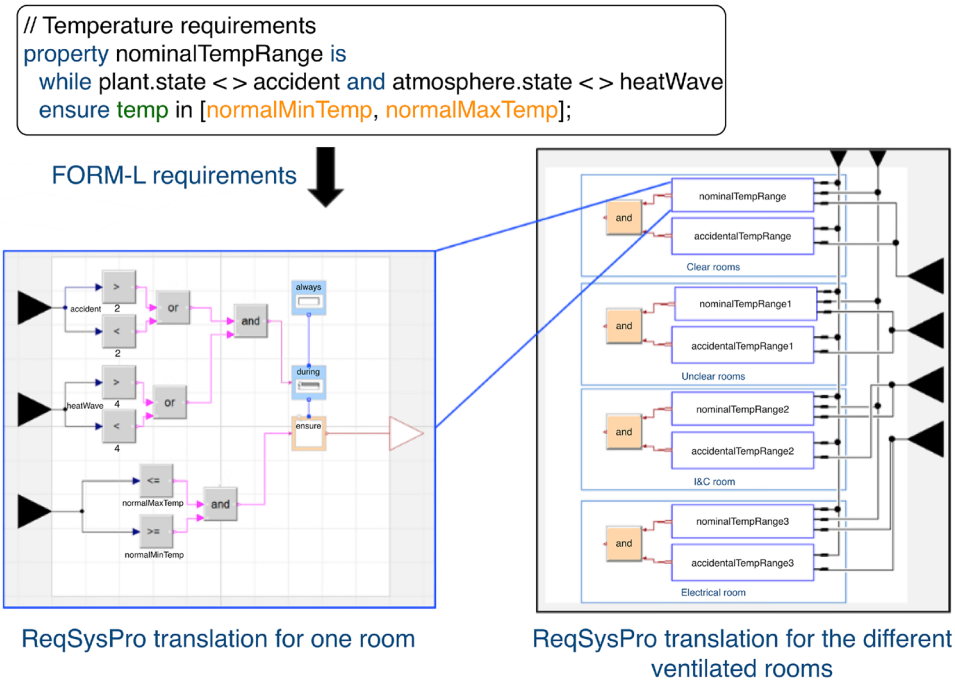

Fig. 18 FORM-L and ReqSysPro translation of HVAC requirement for thermal conditioning

The Trailing-Edge High-Lift System has been defined in the Siemens AMESim simulation platform by using the FORM-L language and the Modelica_Requirements library. The project is released under the LGPL-2.1 License. The source code can be found at the official SMASH-Lab GitHub repository[3], together with the raw simulation results and some scripts to generate the corresponding graphs.

6.2 Case study: the FMEA of a HVAC system

As in critical systems design, verification is as important as the design process itself, performing FMEA in a rigorous way is a key success factor. The goal of FMEA studies is to ensure those component failures will not lead the system to an unacceptable situation. In theory, it should hence drive the engineer to find an optimal system configuration where simple failures will not imply critical consequences. In practice, FMEA studies are today frequently handled manually: (1) engineers list the system components, (2) they identify for each component the possible failure modes, and (3) they assess for each failure model the level of criticality reached when injecting the failure into a static description of the system behavior. Conducting FMEA by hand has several disadvantages. The number of failure modes grows rapidly with the size and the complexity of the system and it is not uncommon to have to process thousands of failure events in industrial systems. Manual evaluation is obviously error-prone and so time-consuming that in reality FMEAs are performed only at the late stages of the design process when all design choices have already been made. FMEA appears then as a duty to substantiate the system safety, but not as an enabler for the designer.

To overcome these difficulties, the requirement verification procedure described in Fig. 4 has been tested to partially automate the FMEA study of an HVAC system. Both the requirements model and dynamic behavioral model have been used. The objective of an HVAC system is to heat and climate a set of rooms "Rooms" and to ensure good air quality for the occupants. Let us focus here only on its thermal conditioning mission.

Informally, the HVAC system has hence a contract $\mathfrak{C}_t(A_i, G_i)$ with each room r_i such as: "in normal or fault conditions (which means with the assumption that room r_i produces a given amount of heat), the HVAC system is designed to guarantee an overall room temperature between *minTempRi* and *maxTempRi*". Such requirement R can be translated into FORM-L as:

```
requirement nominalTempRange is
 for all r_i in Rooms
 while plant.state <> accident and atmosphere.state <>
     heatwave
 ensure tempRi in [minTempRi, maxTempRi];
```

[3] https://github.com/SMASH-Lab/Trailing-Edge-High-Lift-System.

Scenario n°i: Normal operation + component failure n°i	Components	Failure Mode	Critically Level Reached
1	Protection Grid	Fails Blocked (FB)	1
2	Fan	Failure to Start (FS)	2
3	Fan	Failure to Run (FR)	2
4	Fan	Failure to Restart (FRS)	2
5	Fan	Spurious Operation (SO)	1
6	Fan	Run-On (RO)	1
7	Manuel Valve	Valve Left in Wrong Position (EW)	2
8	Manuel Valve	Rupture (RU)	3
9	Cooling Coil	Fails Blocked (FB)	1

Fig. 19 HTML report produced automatically from the different simulations of the verification model

Requirement *nominalTempRange* has then been translated manually into ReqSysPro blocks (Fig. 18). A Modelica model has been used to describe the dynamic behavior of the rooms and of the HVAC system. In particular, it computes the values of the different room temperatures *tempRi* by using the FORM-L keywords, *while* and *ensure*. In this case, for each room r_i, *while* the system is in operation, i.e., its *plant.state* is not equal to *accident* and *atmosphere.state* is not equal to *heatwave*, the *ensure* condition must be verified. Specifically, *ensure* states that the condition must be verified all along the system operation.

A verification model has been manually developed by coupling the Modelica behavioral model with the ReqSysPro model. To do so, a binding model has been added to connect the appropriate variables together (for instance, to convert the temperatures *tempRi* of the behavioral model from Kelvin to Celsius degrees to be coherent with the unit system of the ReqSysPro model). Then, the possible failure modes have been listed in an Excel file (one column per mode), and a kind of "bindings" has been performed to see how each failure event could be emulated in the behavioral model and what is the corresponding scenario to simulate (for instance, a fan failure can be modeled as an air mass flowrate decreasing to zero).

A dedicated Python application has then been developed to: (1) run in series the simulation of the verification model according to the various scenarios listed in the Excel file, (2) retrieve from the verification model the status of each requirement (here to know whether each room temperature is in the authorized range or not) and (3) generate an HTML report in a "FMEA-style" to recap what is the obtained level of criticality (1, 2 or 3) for each failure mode (Fig. 19). This step is a post-processing of the simulation results to be more easily interpreted by the engineer. It is a kind of a dictionary that converts the status of the system requirements (i.e., *satisfied*, *violated*, etc.) in terms of physical impact on the system. This dictionary is set manually as it depends on the system and embeds some physical expertise. Here, criticality level 1 means that the initiating failure does not have any impact on the thermal conditioning, level 2 leads to unavailability of the cooling system but within a time period sufficiently long to allow repairing the default before losing completely the system division, level 3 means that a backup division should be started, and level 4 refers to a critical issue when the complete cooling system is lost.

Such automation brings many advantages. Specifically, it guarantees that the tests performed are exhaustive (no risk of forgetting one case among thousands since the simulations are scripted). It allows a faster (and hence cheaper) way of getting the results. It offers the possibility to realize FMEA studies all along the system lifecycle, as soon as requirements are changed (for instance, when a more stringent law is enacted) or when the design is reviewed (for instance, to test alternatives). This is in particular useful for impact analysis. It helps engineers to assess the design margins to consider and test all the different types of solutions that they can envision when a requirement is violated (should he modify the design or renegotiate the contract?). Temporal aspects can now be considered in FMEA studies by the possible use of dynamic models, which is not feasible by hand. This provides new possibilities to offer additional operational margins or flexibility, or at least a better understanding of failure consequences. Finally, the combination of failure components can be imagined and tested which can be useful to test more stringent safety scenarios with aggravating factors.

7 Discussion and conclusion

To support the design, development and operation of CPS, many research efforts are focusing on the definition of methods, models and techniques capable of dealing, in an integrated way, with the difficulties associated with managing, tracking and verifying system properties, requirements and constraints throughout the CPS lifecycle, through the use of Modeling and Simulation (M and S) techniques.

The paper presented an integrated solution based on the FORM-L and Modelica language that allows formally defining system requirements and automate their verification through simulation by the use of an integrated toolchain. The integrated solution delineates a process of requirement engineering resulting from the integration and extension of research results mainly achieved within the MODRIO project.

The solution is exemplified through two case studies related to the MODRIO project. Concerning the Trailing-Edge High-Lift system, the main objective of the experimentation was the evaluation of different design alternatives and the subsequent identification of the best one that meets the system requirements. As regards the HVAC system, the objective was to automate the FMEA studies in order to be able to rigorously assess the system safety all along the design process, i.e., as soon as modifications are applied to the system design and not only at the end of the design process.

The proposed solution has been experimented and currently adopted by the MODRIO project partners. Some commercial and non-commercial software libraries supporting the method have been developed and are available on the Modelica community repository[4], where a considerable number of downloads have been registered in a few months; this testifies to a great interest in the proposed solution by the various scientific communities that goes beyond the partners of the MODRIO project. Although the proposed solution offers many advantages, it presents some limitations and challenges, such as: (i) the requirements have to be expressed in a rigorous way as delineated in Subsection 4.1; (ii) the FORM-L language is tool independent but, at the moment, it has been formalized only for the Modelica environment; (iii) the conducted experimentations of the proposed solution were limited to the partners involved in the MODRIO project.

To overcome the presented limitations and challenges, future research efforts will be devoted to: (i) improve and extend Modelica libraries to support a wider set of FORM-L concepts; (ii) build FORM-L compilers to generate automatically simulation codes from FORM-L models; (iii) build an Ontology-based component that allows the automatic generation of test sequences starting from a FORM-L model; (iv) investigate the possibility to generate semi-automatically FORM-L models from requirements expressed in natural language; (v) investigate new strategies for automating models bindings and requirements verifications; (vi) perform further experimentations of the solution in different application domains; and, (vii) conduct a user satisfaction campaign through the channels offered by OSMC (Open Source Modelica Consortium)[5] to assess the diffusion of the solution along with its strengths and weaknesses.

Acknowledgements This paper is based on research performed in the context of the MOdel DRiven physical systems Operation (ITEA 3—MODRIO) European Project that aimed at extending modeling and simulation languages and tools based on open standards from system design to system operation. The authors would like to thank Yulu Dong (EDF Lab), Pascal Borel and Felix Marsollier (Edvance) for their valuable contribution on the experiment reported in Subsection 6.2, and Pierre Weis (Inria) and Habib Jreige (Sciworks Technologies) for developing the FORM-L to Modelica compiler.

Funding Open access funding provided by Università della Calabria within the CRUI-CARE Agreement. The authors received no specific funding for this work.

Code availability The Modelica_Requirements library is an open source Modelica library that implements a subset ofthe FORM-L language in form of Modelica blocks. It is available on the modelica-3rdparty GitHubrepository at: https://github.com/modelica-3rdparty/Modelica_Requirements.Examples reported in the paper are available at: https://github.com/SMASH-Lab/Trailing-Edge-High-Lift-System

Declarations

Conflict of interest All authors declare that they have no conflict of interest.

References

1. Broy M, Schmidt A (2014) Challenges in engineering cyber-physical systems. Computer 47(2):70–72

[4] https://github.com/modelica.

[5] https://openmodelica.org/home/consortium.

2. Bocciarelli P, D'Ambrogio A, Falcone A, Garro A, Giglio A (2019) A model-driven approach to enable the simulation of complex systems on distributed architectures. Simul Trans Soc Model Simul Int. https://doi.org/10.1177/0037549719829828
3. Dick J, Hull E, Jackson K (2017) Requirements engineering. Springer, Berlin
4. Cheng BHC, Atlee JM (2007) Research directions in requirements engineering. In: International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA, pp. 285–303. https://doi.org/10.1109/FOSE.2007.17
5. Ericson CA et al (2015) Hazard analysis techniques for system safety. Wiley, Hoboken
6. Garro A, Groß J, Gen Richter MR, Tundis A (2014) Reliability analysis of an attitude determination and control system (ADCS) through the RAMSAS method. J Comput Sci 5(3):439–449. https://doi.org/10.1016/j.jocs.2013.06.003
7. Garro A, Tundis A (2015) On the reliability analysis of systems and SoS: the RAMSAS method and related extensions. IEEE Syst J 9(1):232–241. https://doi.org/10.1109/JSYST.2014.2321617
8. Liu HC, Liu L, Liu N (2013) Risk evaluation approaches in failure mode and effects analysis: A literature review. Expert Syst Appl 40(2):828–838
9. Falcone A, Garro A, Taylor SJE, Anagnostou A, Chaudhry NR, Salah O (2017) Experiences in simplifying distributed simulation: The HLA development kit framework. J Simul 11(3):208–227. https://doi.org/10.1057/s41273-016-0039-4
10. ITEA 3 MODRIO: Model driven physical systems operation project (2018). https://itea3.org/project/modrio.html. Accessed 23 Jun 2018
11. Ramos AL, Ferreira JV, Barceló J (2011) Model-based systems engineering: an emerging approach for modern systems. IEEE Trans Syst Man Cybern Part C (Appl Rev) 42(1):101–111
12. Jardin A, Bouskela D, Nguyen T, Ruel N, Thomas E, Chastanet L, Schoenig R, Loembé S (2011) Modelling of system properties in a Modelica framework. In: Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany, 63, pp. 579–592. Linköping University Electronic Press
13. Wu J, Liu G, Lane V (1999) Formal verification. CIS 841 Web Book
14. Modelica Association: the Modelica association international home page (2018). https://www.modelica.org/. Accessed 23 June 2018
15. Nguyen T (2014) Form-l: A Modelica extension for properties modelling illustrated on a practical example. In: Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden, 096, pp. 1227–1236. Linköping University Electronic Press
16. Aiello F, Garro A, Lemmens Y, Dutré S (2017) Simulation-based verification of system requirements: An integrated solution. In: Proceedings of the 14th IEEE International Conference on Networking, Sensing and Control (ICNSC), pp. 726–731. IEEE
17. Garro A, Tundis A (2015) Modeling of system properties: Research challenges and promising solutions. In: Systems Engineering (ISSE), 2015 IEEE International Symposium on Systems Engineering, pp. 324–331. IEEE
18. Garro A, Tundis A, Bouskela D, Jardin A, Thuy N, Otter M, Buffoni L, Fritzson P, Sjölund M, Schamai W, et al (2016) On formal cyber physical system properties modeling: A new temporal logic language and a modelica-based solution. In: 2016 IEEE International Symposium on Systems Engineering (ISSE), pp. 1–8. IEEE
19. Otter M, Thuy N, Bouskela D, Buffoni L, Elmqvist H, Fritzson P, Garro A, Jardin A, Olsson H, Payelleville M, et al (2015) Formal requirements modeling for simulation-based verification. In: Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015, 118, pp. 625–635. Linköping University Electronic Press
20. Pinquié R, Micouin P, Véron P, Segonds F (2016) Property model methodology: a case study with modelica. Tools and Methods of Competitive Engineering (TMCE)
21. Alur R (2015) Principles of cyber-physical systems. MIT Press, Cambridge
22. Falcone A, Garro A, Tundis A (2014) Modeling and simulation for the performance evaluation of the on-board communication system of a metro train. In: Proceedings of the 13th International Conference on Modeling and Applied Simulation (MAS 2014), Bordeaux (France)
23. Sankaranarayanan S, Sipma HB, Manna Z (2004) Constructing invariants for hybrid systems. In: International Workshop on Hybrid Systems: Computation and Control, pp. 539–554. Springer
24. ISO/IEC/IEEE 15288:2015: ISO/IEC/IEEE 15288:2015. Systems and software engineering: system life cycle processes (2015)
25. Knight JC (2002) Safety critical systems: challenges and directions. In: Proceedings of the 24th international conference on software engineering, pp. 547–550. ACM
26. Ferrari A (2018) Natural language requirements processing: From research to practice. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 536–537
27. Reiterer SH, Balci S, Fu D, Benedikt M, Soppa A, Szczerbicka H (2020) Continuous integration for vehicle simulations. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), vol. 1, pp. 1023–1026. IEEE
28. Ji J, Peng R (2016) An analysis pattern driven requirements modeling method. In: 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), pp. 316–319. https://doi.org/10.1109/REW.2016.058
29. Ledru Y, Blein Y, du Bousquet L, Groz R, Clere A, Bertrand F (2018) Requirements for a trace property language for medical devices. In: 2018 IEEE/ACM International Workshop on Software Engineering in Healthcare Systems (SEHS), pp. 30–33
30. Ferrari A, Spagnolo GO, Gnesi S (2017) Pure: A dataset of public requirements documents. In: 2017 IEEE 25th International Requirements Engineering Conference (RE), pp. 502–505. https://doi.org/10.1109/RE.2017.29
31. Yang K, Zhao Q, Lu Y, Huang W (2009) The research of system of systems requirement modeling and toolkits. In: 2009 11th International Conference on Computer Modelling and Simulation, pp. 107–110. https://doi.org/10.1109/UKSIM.2009.115
32. Zambonelli F, Jennings NR, Wooldridge M (2003) Developing multiagent systems: the gaia methodology. ACM Trans Softw Eng Methodol (TOSEM) 12(3):317–370
33. Buffoni-Rogovchenko L, Fritzson P, Nyberg M, Garro A, Tundis A (2013) Requirement verification and dependency tracing during simulation in modelica. In: Modelling and Simulation (EUROSIM), 2013 8th EUROSIM Congress on, pp. 561–566. IEEE
34. Rogovchenko-Buffoni L, Tundis A, Fritzson P, Garro A (2013) Modeling system requirements in Modelica: Definition and comparison of candidate approaches. In: Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2013, April 19, University of Nottingham, Nottingham, UK, pp. 15–24
35. Fritzson P (2014) Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach. Wiley
36. Open Modelica: The Open Modelica website (2019). https://www.openmodelica.org/. Accessed 20 Dec 2019
37. Bauer A, Leucker M (2011) The Theory and Practice of SALT. In: Bobaru M, Havelund K, Holzmann GJ, Joshi R (eds) NASA Formal Methods. Springer, Berlin Heidelberg, pp 13–40. https://doi.org/10.1007/978-3-642-20398-5_3

38. Object Management Group (OMG): Systems modeling language (SysML) (2019). www.omgsysml.org/. Accessed 20 Dec 2019
39. Behjati R, Yue T, Nejati S, Briand L, Selic B (2011) Extending SysML with AADL concepts for comprehensive system architecture modeling. In: European Conference on Modelling Foundations and Applications, pp. 236–252. Springer
40. de Saqui-Sannes P, Hugues J (2012) Combining SysML and AADL for the design, validation and implementation of critical systems. In: ERTS2 2012, p. 117
41. Vilela J, Castro J, Martins LEG, Gorschek T, Silva C (2017) Specifying safety requirements with gore languages. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, pp. 154–163
42. Joochim T, Poppleton M (2007) Transforming timing diagrams into knowledge acquisition in automated specification
43. Krishna A, Le Pallec M, Mateescu R, Noirie L, SalaÃn G (2019) Iot composer: composition and deployment of IoT applications. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 19–22. https://doi.org/10.1109/ICSE-Companion.2019.00028
44. Shmeleva TR (2017) Automated composition of petri net models for cellular structures. In: 2017 IEEE First Ukraine Conference on Electrical and Computer Engineering (UKRCON), pp. 1019–1024 https://doi.org/10.1109/UKRCON.2017.8100405
45. Zhu W, Bastani F, Yen I, Fu J, Zhang Y (2017) Automated holistic service composition: modeling and composition reasoning techniques. In: 2017 IEEE International Conference on Web Services (ICWS), pp. 596–603. https://doi.org/10.1109/ICWS.2017.70
46. Castellanos C, Borde E, Pautet L, Sébastien G, Vergnaud T (2015) Improving reusability of model transformations by automating their composition. In: 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, pp. 267–274. https://doi.org/10.1109/SEAA.2015.76
47. Maigre R (2010) Survey of the tools for automating service composition. In: 2010 IEEE International Conference on Web Services, pp. 628–629. https://doi.org/10.1109/ICWS.2010.72
48. Schamai W, Buffoni L, Fritzson P (2014) An approach to automated model composition illustrated in the context of design verification. Model Identif Control 35(2):79
49. De Giacomo G, Mecella M, Patrizi F (2014) Automated service composition based on behaviors: The roman model. In: Web Services Foundations, pp. 189–214. https://doi.org/10.1007/978-1-4614-7518-7_8
50. Rosenmüller M, Siegmund N, Apel S, Saake G (2011) Flexible feature binding in software product lines. Autom Softw Eng 18(2):163–197
51. Granda MF, Condori-Fernandez N, Vos TEJ, Pastor O (2017) Costest: A tool for validation of requirements at model level. In: 2017 IEEE 25th International Requirements Engineering Conference (RE), pp. 464–467. https://doi.org/10.1109/RE.2017.69
52. Adedjouma M, Yakymets N (2019) A framework for model-based dependability analysis of cyber-physical systems. In: 2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE), pp. 82–89. https://doi.org/10.1109/HASE.2019.00022
53. Dassault Systèmes: The stimulus project home page (2020). https://www.argosim.com. Accessed 31 Jan 2020
54. Reinbacher T, Rozier KY, Schumann J (2014) Temporal-logic based runtime observer pairs for system health management of real-time systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 357–372. Springer
55. Sippl C, Bock F, Lauer C, Heinz A, Neumayer T, German R (2019) Scenario-based systems engineering: An approach towards automated driving function development. In: 2019 IEEE International Systems Conference (SysCon), pp. 1–8. https://doi.org/10.1109/SYSCON.2019.8836763
56. Balsini A, Di Natale M, Celia M, Tsachouridis V (2017) Generation of Simulink monitors for control applications from formal requirements. In: 2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 1–9. IEEE
57. Mavridou A, Bourbouh H, Giannakopoulou D, Pressburger T, Hejase M, Garoche PL, Schumann J (2020) The Ten Lockheed Martin Cyber-Physical Challenges: formalized, Analyzed, and Explained. In: 2020 IEEE 28th International Requirements Engineering Conference (RE), pp. 300–310. IEEE. https://doi.org/10.1109/RE48521.2020.00040
58. Mustafa N, Labiche Y, Towey D (2018) Traceability in systems engineering: An avionics case study. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 02, pp. 818–823. https://doi.org/10.1109/COMPSAC.2018.10345
59. Jarke M (1998) Requirements tracing. Commun ACM 41(12):32–36
60. Shankar N (2000) Combining theorem proving and model checking through symbolic analysis. In: International Conference on Concurrency Theory, pp. 1–16. Springer
61. Open Services for Lifecycle Collaboration (OSLC): The oslc international home page (2021). https://open-services.net/. Accessed 01 Jan 2021
62. IEEE Std 24748-1-2011: IEEE Guide–Adoption of ISO/IEC TR 24748-1:2010 Systems and Software Engineering–Life Cycle Management–Part 1: Guide for Life Cycle Management (2011). https://doi.org/10.1109/IEEESTD.2011.5871657
63. Bouskela D, Jardin A (2018) ETL: A new temporal language for the verification of cyber-physical systems. In: 2018 Annual IEEE International Systems Conference (SysCon), pp. 1–8. IEEE
64. Benveniste A, Caillaud B, Nickovic D, Passerone R, Raclet JB, Reinkemeier P, Sangiovanni-Vincentelli A, Damm W, Henzinger TA, Larsen KG et al (2018) Contracts for system design. Found Trends® in Electron Des Autom 12(2–3):124–400
65. Bouskela D, Nguyen T, Jardin A (2017) Toward a rigorous approach for verifying cyber-physical systems against requirements. Can J Electr Comput Eng 40(2):66–73
66. Daimler AG: The functional mock-up interface (FMI) standard home page (2019). http://www.fmi-standard.org. Accessed 23 Jun 2019
67. Gomes C, Thule C, Broman D, Larsen PG, Vangheluwe H (2018) Co-simulation: a survey. ACM Comput Surv (CSUR) 51(3):1–33
68. Dal Monte A, Castelli MR, Benini E (2012) A retrospective of high-lift device technology. In: Proceedings of World Academy of Science, Engineering and Technology, 71, p. 1979. World Academy of Science, Engineering and Technology (WASET)
69. Siemens: Siemens amesim-motion simulation environment home page (2019). https://www.plm.automation.siemens.com. Accessed 23 Jun 2019
70. Garro A, Falcone A (2015) On the integration of HLA and FMI for supporting interoperability and reusability in distributed simulation. In: Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, part of the 2015 Spring Simulation Multiconference, SpringSim 2015, Alexandria, VA, USA, April 12-15, 2014, pp. 9–16. The Society for Modeling and Simulation International Inc.. http://dl.acm.org/citation.cfm?id=2872967
71. Falcone A, Garro A (2019) Distributed co-simulation of complex engineered systems by combining the high level architecture and functional mock-up interface. Simul Model Pract Theory. https://doi.org/10.1016/j.simpat.2019.101967
72. Zaccai D, Bertels F, Vos R (2016) Design methodology for trailing-edge high-lift mechanisms. CEAS Aeronaut J 7(4):521–534