Software and Systems Modeling (2024) 23:1455–1491
https://doi.org/10.1007/s10270-024-01228-3

THEME SECTION PAPER

Completion of SysML state machines from Given–When–Then requirements

Maria Stella de Biase[1] · Simona Bernardi[2] · Stefano Marrone[1] · José Merseguer[2] · Angelo Palladino[3]

Received: 14 August 2023 / Revised: 16 September 2024 / Accepted: 2 October 2024 / Published online: 27 November 2024

Abstract
MDE enables the centrality of the models in semi-automated development processes. However, its level of usage in industrial settings is still not adequate for the benefits MDE can introduce. This paper proposes a semi-automatic approach for the completion of high-level models in the lifecycle of critical systems, which exhibit an event-driven behaviour. The proposal suggests a specification guideline that starts from a partial SysML model of a system and on a set of requirements, expressed in the well-known Given–When–Then paradigm. On the basis of such requirements, the approach enables the semi-automatic generation of new SysML state machines model elements. Accordingly, the approach focuses on the completion of the state machines by adding proper transitions (with triggers, guards and effects) among pre-existing states. Also, traceability modelling elements are added to the model. Two case studies demonstrate the feasibility of the proposed approach.

Keywords Behaviour-driven development · Requirements engineering · SysML · Critical systems design · Event-driven systems design

1 Introduction

Model-driven development is a research field that advocates the use of models as first-class citizens in the software development process [9, 22]. A primary interest is the automatic creation of software artefacts (including models) from models. Among the latter, behavioural models play a prominent role, for many important reasons. For example, they are fundamental for carrying out all kinds of non-functional analyses, e.g. security [4], performance [15] or dependability [6, 7], also behavioural specifications provide the means for executable specifications [49]. Behaviour-driven development (BDD),[1] as an evolution of Test-driven development (TDD) [5], elevates behavioural specifications to guide development, system testing and communication with stakeholders.

Behavioural models, such as the System Modelling Language (SysML) [40] or the Unified Modelling Language (UML) [39] state machines, are often created manually, which is a costly and error-prone task. However, we are confident in taking advantage of much of the information gathered during the early stages of the development process, mainly requirements and architectural diagrams. In our

Communicated by Lano, Kolahdouz-Rahimi, Yassipour-Tehrani, Burgueño and Uma.

✉ Simona Bernardi
simonab@unizar.es

Maria Stella de Biase
mariastella.debiase@unicampania.it

Stefano Marrone
stefano.marrone@unicampania.it

José Merseguer
jmerse@unizar.es

Angelo Palladino
angelo.palladino@kineton.it

1 Dipartimento di Matematica e Fisica, Università della Campania “Luigi Vanvitelli”, viale Lincoln, 5, 81100 Caserta, Italy

2 Department de Informática e Ingeniería de Sistemas Escuela de Ingeniería y Arquitectura, Universidad de Zaragoza, Edificio Ada Byron Calle María de Luna 1, 50018 Zaragoza, Spain

3 Aerospace Business Unit, Kineton SRL, via Emanuele Gianturco, 23, 80146 Napoli, Italy

1 https://dannorth.net/introducing-bdd/.

view, all these models and information may help to create behavioural models automatically.

Because the automatic creation of behavioural models is highly challenging, this work starts exploring autocompletion as an initial step in that direction. Model autocompletion was introduced by Burgueño et al. [12] as a new feature for a future generation of modelling tools. In their opinion, autocompletion could significantly improve the modelling task and also modelling assistant tools [38], such as OutSystems [41] or Mendix.[2]

This paper contributes to two goals. As a first goal, it leverages SysML block diagrams and class diagrams to semicomplete SysML state machines. Hence, proposing an approach that identifies patterns in the system requirements (expressed using Gherkin [53], as usual in BDD), so to assist the autocompletion process. As a second goal, the paper illustrates these findings in two case studies. The first one in European Rail Traffic Management System (ERTMS)/European Train Control System (ETCS). The second one, proposed by Douglass in [18], considers a specific critical event in the healthcare domain.

This work builds on previous results, described in the preprint [8] where a preliminary idea was sketched. Herein, in pursuing the mentioned goals, our approach entails the following original contributions:

– Definition of a mechanism for a dynamically adapted grammar, so to analyse system requirements;
– Definition of a knowledge base of possible requirement schemes and related SysML model adaptations;
– Creation of a toolchain, open to integrate Natural Language Processing (NLP) techniques that improve flexibility.

The scope of this work is in the field of event-driven systems, which are usually modelled using State Machines. On the other hand, the proposed approach requires modelling with SysML, the usage of some requirements guidelines and requirements traceability and all these characteristics often found in critical systems. Certainly, both case studies, nicely represent an intersection between these two categories of systems, the event-driven ones and the critical ones. The first system using a messaging between vital ends in the ERTMS/ETCS specification. The second system focuses on the design of a system to detect and respond to critical events in an operational environment.

The rest of the paper is organized as follows. Section 2 introduces necessary background, and Sect. 3 reviews related works. Section 4 explains the proposed methodology. The next sections define the technical details. Section 5 reports on modelling. Sections 6, 7 and 8 report on knowledge representation. Section 9 describes a supporting process and toolchain. The previous sections use a software controlled lamp as a running example. Section 10 illustrates the approach with a case study based on ETCS Level 3 (ETCS-L3). Section 11 evaluates the approach using the healthcare case study. The paper concludes with remarks and observations in Sect. 12.

2 Background

This section introduces basic concepts that help to focus on the context of the paper.

2.1 Behaviour-driven development

BDD is a software development methodology that aims to improve communication between all involved stakeholders through the formulation of use and test scenarios [49]. BDD focuses on the behaviours that the system should have, rather than just on the desired functionality. Thus, the system behaviours must be clearly and unambiguously defined through *Scenarios*—presented in Sect. 3. The Gherkin language is mainly associated with BDD, and it employs a plain-text format to define the expected behaviour of the software system. Gherkin is typically implemented in combination with frameworks and tools, such as Cucumber [53], and allows the definition of scenarios, which are described in a simple, non-redundant and understandable way by different teams, without source of ambiguity. Scenarios, in addition to describing a behaviour or functionality, are executable specifications, i.e. they define the skeleton of acceptance tests that are useful during the system verification phase. They, therefore, help to ensure the correct behaviour of the system.

Gherkin was chosen for this work primarily because of one powerful characteristic: the linguistic structure's simplicity (Given–When–Then) combined with its capability for verification. The authors in [2] highlight:

> The popularity of the Gherkin language is, in large part, due to its ability to enforce the use of high-level, domain-specific terms and supporting traceability from AC (acceptance criteria) to executable test cases. More precisely, the Gherkin syntax enables the automated generation of executable test cases based on matching AC text to APIs, thus leading to test case traceability.

2.2 ERTMS/ETCS-L3

The standard ERTMS has been designed to improve efficiency, interoperability and safety in railway operations [21]. In particular, ERTMS establishes mandatory specifications

[2] https://www.mendix.com/platform/#assist.

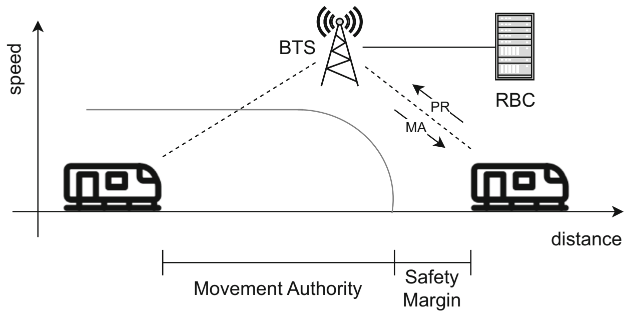


Fig. 1 The ETCS control reference schema

on Trans European Network Routes. The main components of the ERTMS are (1) European Traffic Management Layer (ETML); (2) ETCS, in charge of the movement of the train; and (3) Global System for Mobile Communications for Railways (GSM-R), responsible for the radio communication.

Different uses of ERTMS—resulting in different performance system profiles—are made possible on the basis of the selected technologies. In particular, ETCS considers three levels—defining how trackside, lineside and on-board controllers are related to each other—which play a crucial role in the determination of system overall performance.

The first two levels—i.e. ETCS Level 1 (ETCS-L1) and ETCS Level 2 (ETCS-L2)—use fixed block signalling, i.e. are based on "adding up" hardware equipment to better control the train running and the delivery of the Movement Authority (MA), which authorizes a train to run safely for a certain space. In particular, in ETCS-L2, the calculation of such distance is computed by the Radio Block Centre (RBC) which receives the position of the train via radio message (i.e. the Position Report (PR)) and sends the authorization with another message (i.e. the MA[3]). This situation is depicted in Fig. 1.

One of the future challenges of the railway industry is constituted by increasing the movement of goods on railways, which can be pursued by increasing network capacity (and not only train speed). Thus, extending the railway infrastructure is not always feasible due to the high costs of extra track construction, especially in populous territories [24]. ETCS-L3 introduces moving block as an answer to the problem of "adding" new hardware; in ETCS-L3 the role played by determining the exact position of a train by the train itself is crucial not only for performance but also for safety purposes.

3 Related Work

Scenarios and requirements, when expressed in Natural Language (NL), pose advantages, especially in the industrial setting, since this is an easy way for engineers to communicate with stakeholders [30]. Consequently, the research around Requirement Engineering (RE) and NL has advanced remarkably, as surveyed in [56] and in [31], the latter to include literature on Machine Learning (ML) for NLP. These advances span many heterogeneous fields, from UML specifications [32, 43] to temporal logic formulae [11, 13, 23] through grammar-based approaches [19]. However, there are still many open challenges in the field of RE and NL, as investigated in [16] and in [31]. The latter identifies the need of introducing deep learning techniques in the field, since currently classical ML are predominant, and also the difficulty of comparing different approaches due to the lack of benchmark cases.

More specifically, Controlled Natural Language (CNL) techniques [33] provide templates, such as those of the Gherkin language [53], that assist in capturing requirements while describing BDD scenarios. BDD, as mentioned above, builds upon TDD, and there is extensive literature that automatically produces tests from BDD specifications, e.g. [29]. The Gherkin language defines the desired behaviour of the software system in plain-text format. Listing 1 shows the Gherkin syntax format, also called *Scenario*. Therefore, through the use of the Gherkin Scenario, it is possible to detail every behaviour of the system. In particular, every Scenario is made up of special keywords—some are optional and others are mandatory—the three initial keywords represent different information: (1) *Feature*, which briefly expresses the system characteristic to be defined, it can be used to group related Scenarios; (2) *Rule*, an optional keyword that serves as a representation of a single business rule that needs to be followed; and (3) *Scenario*, a description of the Scenario itself, here, the behaviour of the system—that the Scenario wants to introduce—can be expressed extensively and using NL.

The Gherkin syntax requires, after the description of the Scenario keyword, three mandatory and fundamental keywords: (1) *Given*, (2) *When* and (3) *Then*. The *Given* keyword describes the initial context of the Scenario. The Given sentence is designed to identify the system in a known state before possible interactions between the system and the user or other systems. The *When* sentence represents an event or an action—for example, an action can be a user interacting with the system, or an event can be the receiving of a message from another system. The last keyword, the *Then* one, expresses the desired final state of the system after the event, depicted by the previous keyword, has occurred.

Listing 1 Gherkin Scenario

```
Feature: <<software feature>>
    Rule: <<business rule to implement
        >>
         Scenario: <<description>>
```

[3] In this paper, the term MA both denotes the safe distance and the radio message.

```
Given <<an initial context
    >>
When  <<an event>>
Then  <<an expected outcome
    >>
```

The Gherkin language, in combination with the Cucumber tool, has been of primary importance in most of these test automation processes [36, 53], even in the field of formal methods to produce and execute acceptance tests [50]. Thus, BDD and TDD are very mature fields regarding test automation, and its benefits, challenges and tools have been reviewed in [34, 42]. However, to automatically produce behavioural specifications from requirements, e.g. in Gherkin, is still a task which requires mature scientific and technical proposals.

Reviewing the BDD literature to better learn how to leverage the Gherkin language to produce software artefacts, we have found works in different domains. In the web domain, Dimanidis et al. [17] aim at producing RESTful [44] web services from Gherkin templates. Rocha [45] develops, based on Gherkin, a domain-specific language to specify interaction scenarios to produce web-based graphical user interfaces. Similar to Rocha, Hesenius et al. [25] create a language, based on Gherkin, in this case to test multimodal applications. The fact that both, the application domain and the type of artefacts created by these three approaches, are so different from ours makes it difficult to reuse these approaches in our context, which aims to autocomplete models.

In the automotive domain, Wiecher et al. [52] propose a scenarios-in-the-loop approach to automatically execute and analyse requirements expressed in Gherkin. The ultimate goal is to validate the implemented system through automated tests, but not to create behavioural models as in our approach.

In the field of autocompleting models, where our work also places, we have found the work of Burgueño et al. [12]. They propose an architecture based on NLP to autocomplete domain models, like class diagrams, instead of behavioural models as we do. Then, the automatic analysis of available textual project and historical information provides model autocompletion suggestions. The work, although not in the BDD field and taking a different approach to ours, leverages state-of-the-art NLP techniques instead of patterns. Also, the work in [47] transforms NL requirements into domain models, in this case using machine learning techniques.

Among the works that transform requirements into design models, we have found the one from Colombo et al. [14], that uses SysML and "transformation rules", as in our work. However, in their approach the transformation is entirely model driven, providing transformation rules in ATL [28], while our approach uses a grammar to generate fragments in the state machine semi-automatically. Moreover, our approach uses the Gherkin syntax to express the requirements. Finally, this work considers architecture aspects (data and communication), whereas we focus on system behaviour.

The work of Bruel et al. [10] discusses the effectiveness of different existing approaches to specify requirements. First, the work classifies the approaches in five categories: natural-language, semi-formal, automata/graphs, mathematical and seamless (programming language-based). The authors chose 22 different representative languages of these categories, e.g. Relax [51] for natural-language or Petri nets [37] for automata/graphs. The effectiveness of these categories and languages is assessed according to nine criteria: system vs. environment, audience, level of abstraction, associated method, traceability support, non-functional requirements support, semantic definition, tool support and verifiability. The most salient criteria regarding to formalize requirements are, in our opinion, audience, traceability support, semantic definition, tool support and verifiability. Regarding *audience*, the approaches based on NL are the only ones that do not require the use of special background, hence addressing a larger audience. *Traceability support* is offered by Relax [51] and NL2OWL [35] among NL approaches; however, semi-formal approaches are the best for this criterion. *Semantic definition* is the strong point for automata/graph, mathematical and seamless, which are also the categories best scoring in *verifiability*, but some NL approaches also score in this criterion, e.g. Relax, Stimulus, NL2OCL [27] and NL2STD [1]. Regarding *tool support* almost all the approaches are supported except for NL2OWL and NL2STD.

4 Methodology

The approach proposed aims to support system analysts in the early stages of development. In particular, starting from a preliminary system model, the approach semi-automatically completes the specification of the architecture, leveraging a set of requirements. Figure 2 depicts, at-a-glance, the conceptual architecture of the methodology, which relies upon three layers: User, Knowledge and Tool.

User layer. The *System Analyst*, as responsible for this layer, must provide a preliminary SysML model, which is made of three views.

The *Requirements* view. Requirements can be either expressed using the SysML's Requirement Diagram (RD) or a textual notation. In any case, they need to comply with a notation, detailed in Sect. 6.

The *Structural* and *Behavioural* views. They specify the system architecture. The former can be a SysML's Block Definition Diagram (BD).[4] The latter includes a set of State

[4] Since BDD is used as an acronym for Behaviour-driven development, we prefer to use BD for Block Definition Diagram.

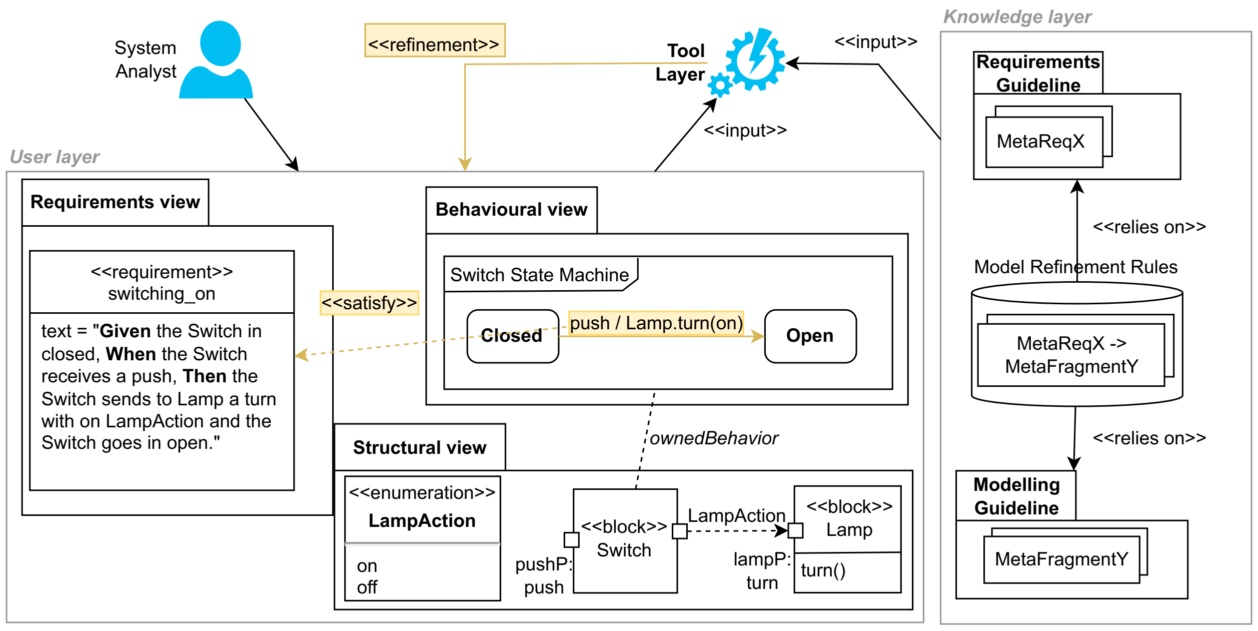


Fig. 2 A conceptual architecture

Machine Diagrams (SMDs), which represent preliminary *owned behaviours* of the blocks in the BD. By preliminary, we mean that only states are defined in the SMD (i.e. white parts in the Behavioural view of Fig. 2).

Knowledge layer. It represents the knowledge base, that is leveraged by the *Tool layer* to semi-automatically autocomplete the preliminary model. This autocompletion process carries out a *detect-and-translate* approach. Concretely, requirement patterns are detected in the Requirements view and then transformed into model fragments in the Behavioural view. A model fragment can be a single model element—i.e. a state machine transition—or a set of model elements—i.e. a transition with a trigger and an effect. Furthermore, traceability information can be added to the model using a <<satisfy>> relationship stereotype from the added model fragment to the generating requirement.

This layer includes two guidelines—*Requirements* and *Modelling*—and a set of *Model Refinement Rules* that rely on the former. The Requirements guideline includes requirement patterns (*MetaReqX*,…), i.e. requirement templates that conform to the Given–When–Then structure. The *Modelling* guideline consists of a set of meta-fragments (*MetaFragmentY*,…), where each one is related to a requirement pattern and it is defined in terms of the roles played by SysML model elements in the requirement pattern. The model refinement rules are aimed at matching a requirement pattern, given a concrete requirement in the Requirement view, and refining the User Layer model by adding new model elements.

From the System Analyst's perspective, the two guidelines support him/her in the creation of the SysML model.

Tool layer. The tool implements the proposed approach. It needs as input both, the SysML model of the system, from the User layer, and the Model Refinement Rules, from the Knowledge layer. Then, the tool instantiates the meta-fragment with the elements of the SysML model and defines the traceability information.

Figure 3 summarizes the workflow implemented by the tool, as follows.

1. *Model Element Extractor* is the first phase of the workflow. It retrieves lists of the element names of the SysML model which are known a-priori. In such named entity lists, there are the names of blocks, states, signals, properties, operations, events and data types, which are all model elements considered in the Knowledge Layer.
2. During the *Processor Building* phase, the named entities already computed, in the previous step, are merged with a model-independent
   Extended Backus-Naur Form (EBNF) grammar. The latter defines the production rules describing the requirement patterns.
   Then, a parser is generated from the resulting model-specific grammar.
3. For each requirement r in the Requirements view, the *Processing* phase is carried out:
   (a) r is parsed and an Abstract Syntax Tree (AST) is generated (*Parsing*);
   (b) The AST is explored and the model refinement rule that matches with the requirement is detected (*Rule Matching*);

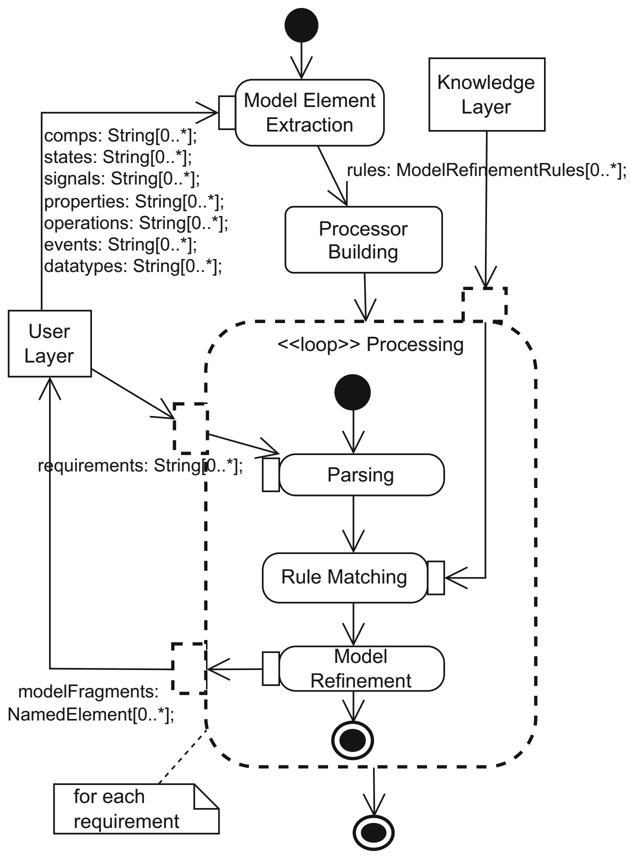


Fig. 3 Tool layer workflow

(c) The matched rule determines the model fragment to apply, and the *Model Refinement* phase is enacted.

5 User Layer

As described in Sect. 4, methodological overview, the SysML model is structured into three views.

Requirements View. This is the simplest view considered in this work, whose aim is to report into SysML's classes tagged with the <<requirement>> stereotype, the text of the system requirements, formatted according to the requirement patterns described in Sect. 6. The requirements can be collected into a SysML's RD, where allocation information is reported, related to the mapping between the requirements and which part of the model fulfils each requirement. This mapping can also take the form of a Requirement Allocation Table (RAT).

Structural View. This is a component-based view of the system in terms of a SysML's BD/Internal Block Diagram (IBD). In such a view, blocks represent physical or software components. Blocks communicate with each other through ports and signals. Ports can be Output Ports and Input Ports according to the direction of the item flows that connect two blocks. Input Ports—e.g. target ports of the item flows—are characterized by a SysML's signal. Item flows are characterized by a property conveyed onto them. This property represents the type of the data transported by the flow.

Behavioural View. Each component of the structural view has a SMD associated, which models its behaviour. We assume that the states of the SMD are known by the system analyst, whereas the transitions need to be semi-automatically created according to the specified requirements. In our modelling approach, signals and operations link the Structural and the Behavioural views via signal and call events, respectively. Considering the interaction via signal events, in the Structural view, the sending and receiving components are characterized by an output and input port, respectively, of type "Signal". The two ports are connected by an item flow which conveys the signal property. In the Behavioural view, the SMD of the sending component can include one or more transitions with effect the generation of a signal event and the SMD of the receiving component can include one or more transitions triggered by a signal event of type "Signal". Alternatively, the interaction can be carried out via call events; in such a case, in the Structural view, an operation is defined in the block representing the callee component, and the item flow between the caller and the callee conveys the operation parameter types. In this view the operation call is modelled as the effect of a transition in the SMD of the caller component, where the actual parameters of the operation conform to the property of the item flow. In the SMD of the callee component, the operation call is modelled as a trigger call event of one or more transitions.

Modelling the running example. The User Layer in Fig. 2 represents a SysML model of a switch and a lamp, as follows.

– The BDD requirement view, with the *switching_on* requirement related to the behaviour of the Switch block;
– The structural view, specified by a BD, that consists of two blocks: *Switch*, which is the source of an item flow to the *Lamp* block. The *Switch* receives *push* signals via its input port *pushP* and interacts with the *Lamp* via *turn()* operation calls. In particular, *turn()* operation has an input parameter of enumeration type *LampAction*—i.e. on/off values—and the item flow between the two blocks conveys the *LampAction* property;
– The SMD represents the behaviour of the *Switch* and, in the preliminary model, only the two states *Closed* and *Open* are defined. The transition between them, together with its trigger (*push*) and effect (i.e. the operation call *Lamp.turn(on)*) is automatically generated from the requirement by following the proposed methodology.

Table 1 Given–When–Then clause templates

Given	
G1	a <<Block as gContext>> in <<State as gSource>>
G2	a <<Block as gContext>>
When	
W1	the <<Block as wReceiver>> receives <<Signal as wEvent>>
W2	the <<Block as wContext>> <<Operation as wAction>>
W3	the <<Property as wVar>> of <<Block as wContext>> <<comp_op>> <<ValueSpecification as wValue>>
W4	the <<Block as receiver>> receives an <<Signal as wEvent>> with <<Property as wVar>>
W5	the <<Property as wVar>> of <<Block as wContext>> <<comp_op>> <<Property as wVar2>> of <<Block as wContext2>>/<<Signal as wContext3>>
W6	the <<Property as wVar>> of <<Block as wContext>> <<comp_op>> <<alias>>
W7	the <<Block as wReceiver>> receives an <<Signal as wEvent>> with <<Property as wVar>> as <<alias>>
Then	
T1	the <<Block as tContext>> <<Operation as tAction>>
T2	the <<Block as tContext>> goes into <<State as tTarget>>
T3	the <<Block as tSender>> sends to <<Block as tReceiver>> a <<Signal as tMessage>> with <<ValueSpecification as tValue>> <<DataType as tType>>
T4	the <<Block as tSetter>> sets the <<Property as tVar>> of the <<Block as tVariableowner>> to <<ValueSpecification as tValue>>
T5	the <<Block as tSetter>> sets the <<Property as tVar>> of the <<Block as tVarOwner>> to <<Property as tVar2>> of <<Block as tVarOwner2>>
T6	the <<Block as tSetter>> sets the <<Property as tVar>> of the <<Block as tVarOwner>> to <<alias>>
T7	the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
T8	the <<Block as tCaller>> calls the <<CallEvent as tCall>> of <<Block as tCalled>>
T9	<<Block as tCaller>> calls the <<CallEvent as tCall>> of <<Block as tCalled>> with <<ValueSpecification as tValue>> <<DataType as tType>>

6 Requirements Guideline

The Requirements guideline, in the Knowledge layer (Fig. 2), offers a set of requirements patterns. A requirement pattern consists of a triplet <G,W,T>. Each triplet is a logical combination (and, or, not) of clause templates (Given, When, Then). Table 1 reports the allowed clause templates.

Clause templates are characterized by <<*metaclass as role*>> fields. *Metaclass* indicates a SysML metaclass (e.g. Block), while *role* refers the role played by this metaclass in the template. In Table 1, <<comp_op>>—in clauses W3, W5 and W6—means any generic comparison operator, e.g. $\geq$, $\leq$, $\neq$, =. It should be noted that not all combinations of clause templates are allowed, as some of them may contain inconsistencies, e.g.:

- a triplet <G,W,T2 and T2> with repetition of T2 template in the Then clause, each with different target states;
- using an alias (as in T6) without having defined it (as example with a proper W6 clause template).

Clause templates are coded into proper production rules of the EBNF grammar in Listing 2. As an example, T2 clause template in Table 1 is defined in the grammar at line 36 of Listing 2. The EBNF grammar defines the requirements patterns without freezing them to a specific model. In particular, the BDD and clause keywords (lines 5–10), the requirement pattern structure (lines 22–26) and clause template structure (lines 27–37) are model independent.

Listing 2 MetaReq grammar

```
1  Helpers
2    ...
3
4  Tokens
5    when_tok = 'when' | 'When';
6    given_tok = 'given' | 'Given';
7    then_tok = 'then' | 'Then';
8    ...
9    receives_tok = 'receives';
10   goes_tok = 'goes';
11   ...
12   <<comp_toks>>
13   <<state_toks>>
14   <<signal_toks>>
15   <<operation_toks>>
16   <<property_toks>>
17   <<event_toks>>
18   <<datatype_toks>>
19   ...
20
21 Productions
22   reqs =
23     {req} req semicolon |
24     {list} req semicolon reqs;
25   req =
26     {req} given [comma1]:comma when
           [comma2]:comma then;
27   given = ...
28   when = ...
29   then =
30     {simple} then_tok then_body;
31   then_body =
32     {unary} then_single |
33     {binary} [then_one]:then_single
           bin_op [then_two]:
          then_single;
34   then_single =
35     {then_case1} article_tok
          block_name operation_name |
36     {then_case2} article_tok
          block_name goes_tok in_tok
          state_name |
37     {then_case3} [art1]:article_tok
           [sender]:block_name
          sends_tok to_tok [art2]:
          article_tok? [receiver]:
          block_name [art3]:
          article_tok signal_name
          with_tok? with_not_tok?
          datatype_name |
38     ...
39   <<comp_rules>>
40   <<state_rules>>
41   <<signal_rules>>
42   <<operation_rules>>
43   <<property_rules>>
44   <<event_rules>>
45   <<datatype_rules>>
46   ...
```

6.1 User Layer knowledge enrichment

The EBNF grammar template in Listing 2 needs to be enriched with information from the User Layer. In order to make it workable for a specific model.

Concretely, those parts enclosed by "<<" and ">>"—lines 12–18 and lines 39–45—are considered placeholders. Hence, they need to be substituted with model-specific information. Hence, during the *Model Element Extraction* and *Processor Building* phases, see Fig. 3, the lists of elements in the SysML model are obtained from the User Layer, then formatted according to the concrete EBNF grammar and finally substituted in the grammar template.

Considering the running example, the list of block names—the `comps: String[0..*];` from the SysML model, `Switch, Lamp`—is transformed into two strings and substituted in two points of the grammar:

- <<comp_toks>> at line 12 is substituted by the content of Listing 3;
- <<comp_rules>> at line 39 is substituted by the content of Listing 4.

Listing 3 Component Names

```
comp1 = 'Switch';
comp2 = 'Lamp';
```

Listing 4 Component Rules

```
block_name =
  {block1} comp1 |
  {block2} comp2;
```

The Processor Building is a specific tool supporting this task. It receives model element names from the Model Element Extraction tool, and it leverages a template engine technology, such as Apache Freemarker[5] or Te4j.[6]

7 Modelling Guideline

The Modelling Guideline establishes the way for refining the SysML model. Concretely, how to create transitions among the already predefined states. For this task, a meta-fragment is specified for each requirement pattern defined in Sect. 6.

Let us consider a requirement pattern <G,W,T>. A meta-fragment defines a mapping, between the *roles* of a clause and the SysML model elements, that conforms to the *metaclasses*. Table 2 summarizes the mappings associated with the clause templates of Table 1.

In general, the Given clause determines the source of the transition to add, the When clause determines the triggers and/or the guards of the transition, and the Then clause detects the target state associated with the transition and determines its effects.

Given clause. The Given rows of Table 2 show the mappings associated with the clauses G1 and G2, respectively. The first case indicates both the block framing the context of the requirement and the state associated with the context, and the G2 clause case does not specify the source state, intending as default that the requirement (and hence the transition) can be applied in each state of the context.

When clause. The next seven rows in the table show the mappings associated with the When clause templates:

– Clauses W1, W2 and W4 are translated into transition triggers, respectively; in particular, W1 generates a trigger from a signal (wEvent), whereas W2 generates a trigger from an operation call (wAction). On the other hand, the difference between W1 and W4 is reflected by the trigger, that in the W4 case includes the signal actual parameter (wVar).
– W3 and W5 are translated into transition guards;
– W6 extends W3, considering an alias, i.e. a variable defined at SysML level as a local property of the state machine or ,in case of the presence of the Modelling and Analysis of Real-Time and Embedded Systems (MARTE) profile), according to the Value Specification Language (VSL);
– W7 extends W4 by adding the definition of an alias into the `entry` section of the target state where the transition ends, containing the definition of the alias according to the used notation.

Then clause. Finally, the remaining rows of the table describe the Then clauses: while the T2 clause is related to the target of the generated transition, the others clauses impact the transition effects.

8 Model Refinement Rules

The model refinement rules are aimed at: 1) matching a requirement pattern given a concrete requirement, which is textually specified in the *Requirement view* of the User Layer model, and 2) refining the User Layer model by adding new model elements.

The pseudo-algorithm, in Listing 5 (expressed in a Python-like notation), specifies the procedure implementing the rule matching and model refinement. In particular, the design of the algorithm follows a *detect-and-translate* approach, which is detailed in the following.

A concrete requirement `req` is firstly parsed to get its abstract syntax tree `ast` (line 2). The latter is then used to *detect* a requirement pattern `mtreq` that matches with the concrete requirement (lines 4–7).

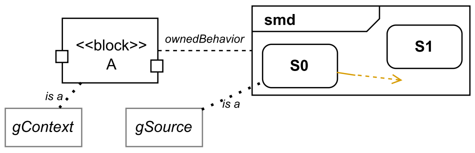
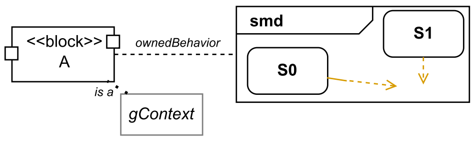
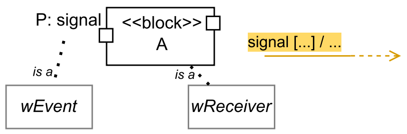
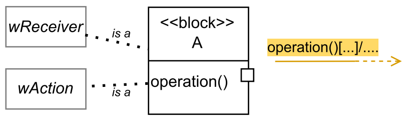
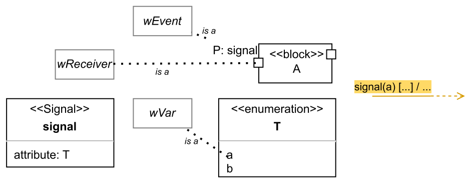
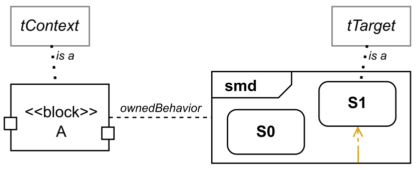
Listing 5 MetaReq2MetaFragment procedure

```
 1  def metareq2metafragment(req,mdl):
 2    ast = parse(req)
 3    # --- Detect ---
 4    mtreq = getMetaRequirement(ast)
 5    isvalid = checkFeasibility(mtreq)
 6    isvalid &= checkRoles(mtreq)
 7    isvalid &= mtreq.checkmodel(mdl)
 8    if isvalid:
 9      # --- Translate ---
10      t = Transition()
11      t.source = mtreq.getSource(mdl)
12      t.target = mtreq.getTarget(mdl)
13      t.trigger = mtreq.getTrigger(
          mdl)
14      t.guard = mtreq.getGuard(mdl)
15      t.effect = mtreq.getEffect(mdl)
16      mdl.add(t)
17      reqblock = mdl.getRequirement(
          req)
18      traceinfo = trace(reqblock,t)
19      mdl.add(traceinfo)
20    return mdl
```

5 https://freemarker.apache.org/index.html.

6 https://github.com/whilein/te4j.

Table 2 Role-to-SysML model element mappings associated with clause templates of Table 1

Given	G1	 smd S1 S0 <<block>> A ownedBehavior is a is a gContext gSource
	G2	 smd S1 S0 <<block>> A ownedBehavior is a gContext
When	W1	 P: signal <<block>> A signal [...] / ... is a is a wEvent wReceiver
	W2	 wReceiver is a <<block>> A operation()[...]/.... wAction is a operation()
	W3	`[wContext.wVar «comp_op» wValue]`
	W4	 wEvent is a P: signal <<block>> A wReceiver is a signal(a) [...] / ... <<Signal>> signal attribute: T wVar is a <<enumeration>> T a b
	W5	`[wContext.wVar «comp_op» wContext2.wVar2]`
	W6	W3 extension considering «alias»
	W7	W4 extension with «alias» definition
Then	T1	`/Activity: tContext.tAction()`
	T2	 tContext is a tTarget is a <<block>> A ownedBehavior smd S1 S0
	T3	`/Activity: send tSender.tMessage(tValue)`
	T4	`/tVariableOwner.tVar = tValue`
	T5	`/tVariableOwner.tVar = tVariableOwner2.tVar2`
	T6	`/tVariableOwner.tVar = alias`
	T7	`/Activity: tSender.tMessage`
	T8	`/Activity: tCalled.tCall()`
	T9	`/Activity: tCalled.tCall(tValue)`

The matching problem is tackled in four steps: first, a candidate requirement pattern `mtreq` is identified (line 4); second, its feasibility is checked (line 5). Indeed, `mtreq` should be structured as a triplet <G,W,T> consisting of a legal combination of Given, When and Then clause templates, as observed in Sect. 6. Third, `mtreq` is checked against a possible mismatching of roles in the matched clause templates (line 6). A mismatching may occur, for example, when `mtreq` is identified as the legal triplet <G1,W1,T2>, but the roles <<gContext>> (in G1) and <<tContext>> (in T2) are different.

Finally, a check of the semantic consistency of `mtreq` regarding the User Layer model `mdl` is done (line 7). The checking is carried out by using the mapping defined by the meta-fragment associated with the detected requirement pattern. In particular, this checking aims at solving the matching between the SysML model and the detected requirement pattern; in fact, some of the matched roles in the clause templates in `mtreq` —which syntactically belong to the correct SysML's metaclass, according to the mechanism of the model-aware grammar—can violate the way the SysML model itself is built. For example, let `mtreq` be the legal triplet <G1,W1,T2>, where "a <<Block as gContext>> in <<State as gSource>>" (G1) does not imply that the gContext's Block has an associated SMD with a state named gSource.

In case the detection of the requirement pattern succeeds, then the *translation* phase follows (lines 9–19), where the User Layer model is possibly refined by adding new model elements.

A candidate transition `t` is firstly created (line 10) and, then, `t` is refined with references to model elements—like, source and target states—named in the detected requirement pattern (lines 11–15), and it is added to the model `mdl` (line 16). Also, `mdl` is enriched with traceability information between the requirement, specified in the Requirement view of the model, and the newly added transition (lines 17–19).

Discussion of translation strategies

In the design of the pseudo-algorithm (Listing 5) a pessimistic strategy is adopted: in fact, the translation phase is not performed when the set of checks (lines 5–7) does not succeed. However, other strategies could be followed where the translation phase is always present, as:

- optimistic: the transition is partially generated from the parts of the requirement that are correct and missing values are present (e.g. guards, triggers);
- default: the missing transition parts are substituted with default arguments.

The strategy can be chosen by the System Analyst, and it should be set according to the level of automation to pursue. As an example, for critical application, the risk of using an optimistic or a default strategy is not negligible; these two possibilities could, in fact, bring to inconsistent model, challenging to debug later. On the other hand, using an optimistic approach could be useful in the case of a set of requirements at their very early stage of writing, since a throw-away model could improve the level of understanding and allow a rewriting of the same requirement set. In any case, detected violations should be signalled to the System Analyst with proper warnings.

Application to the running example.

Let us consider the *switching_on* requirement of the running example, of which Fig. 2 depicts the three views presented in Sect. 5. Table 3 shows the application of the Knowledge Layer matching concepts to the requirement.

In the upper part, the requirement (left side) and the matching MetaReq (right side) are reported. This matching is possible after considering the lower part of the table. Concretely, the lower part shows the terminal symbols (left side) present in the model-specific grammar—defined according to the techniques described in Sect. 6—matching the list of the roles defined in the fields of the MetaReq (right side).

Once the matching is done, and the verification of the above-mentioned constraints passes, the model fragment is generated, refining the SysML model by adding the yellow part of the User Layer in Fig. 2.[7].

9 Tool Layer

This section reports a reference architecture for the Tool Layer proposed in Fig. 2. This architecture, based on the content of Sects. 5–8, is able to automate the proposed methodology.

The components and artefacts of the architecture are grouped in three sets, see Fig. 4. At the centre of the architecture, a master control component, the *Core*, orchestrates and schedules the proper sequence of service invocations.

The *Model Processing* is an interface for reading and writing the SysML model elements. It groups a component, the *Model Handler*, and an <<artifact>>, the *SysML Model*.

The *Parser Generator* analyses the models, hence it implements the phases Model Element Extraction and Processor Building in Fig. 3, as follows:

- The *Processor Builder*, which retrieves the lists of SysML metaclasses, according to the substitution mechanism reported in Sect. 6, detailed in Subsection 6.1. They are substituted in the *Grammar Template* to generate the *Model-Specific Grammar*.
- The *Parser Generator*, which uses the Model-Specific Grammar to generate a parser for this EBNF grammar. Practically, a set of software classes are generated (i.e. the *Parser*), able to be integrated with the rest of the software components. Many parser generators are available for this task, e.g. SableCC.[8]

[7] In the figure, traceability is simplified by a dependency across two views As it is reported in the case study, a dedicated diagram—an RD —best shows such dependencies.

[8] https://sablecc.org/

Table 3 Textual BDD requirement, meta-requirement and elements matching

Requirement	MetaReq (<G1,W1,T3 and T2>)
Given the Switch in closed	Given <<Block as gContext>> in <<State as gSource>>
When the Switch receives push	When <<Block as wContext>> receives <<Signal as wEvent>>
Then the Switch sends to Lamp	Then the <<Block as tSender>> sends to <<Block as tReceiver>>
a turn with on LampAction	a <<Signal as tMessage>> with <<ValueSpecification as tValue>> <<DataType as tType>>
and	and
the Switch goes in open	<<Block as tContext>> goes in <<State as tTarget>>

Role Matching Value	Role
Switch	gContext (Block)
Closed	gSource (State)
Switch	wContext (Block)
push	wEvent (Signal)
Switch	tSender (Block)
Lamp	tReceiver (Block)
turn	tMessage (Signal)
on	tValue (ValueSpecification)
LampAction	tType (DataType)
Switch	tContext (State)
Open	tTarget (State)

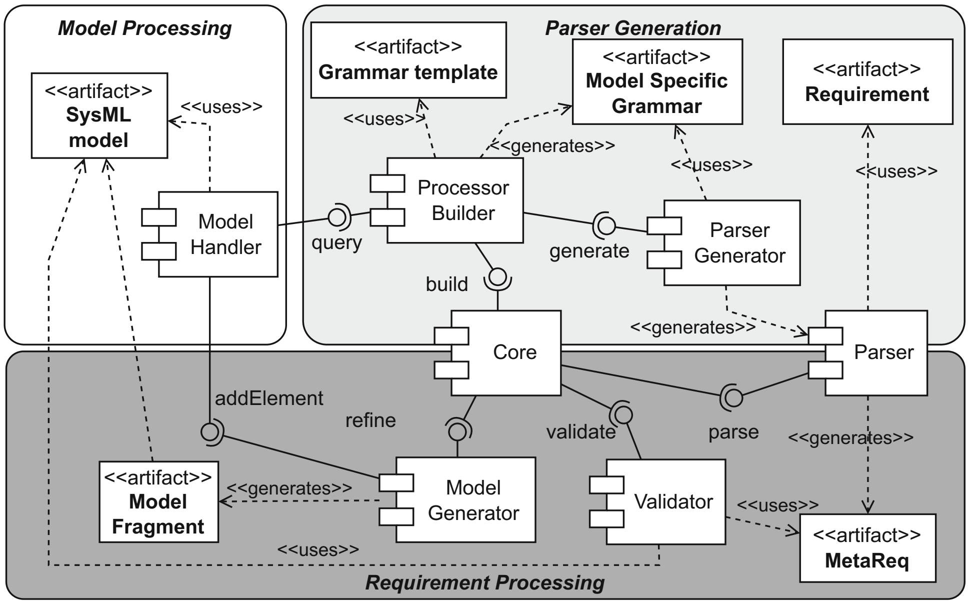


Fig. 4 Tool layer architecture

The *Requirement Processing* automates those phases that iterate each requirement, see Fig. 3. Its three components are:

– The *Parser*, generated in previous stages, that detects the candidate requirement pattern (lines 2 of Listing 5).
– The *Validator* performs a comparison between the meta-requirement (obtained in line 3 of Listing 5) and the formal grammar's rules (lines 4–7 of Listing 5). Only if the validation is successful it is possible to move on to the translation phase.
– The *Model Generator*, that is able to generate the model fragment to add to the SysML model (lines 11–19 of Listing 5).

10 The ETCS-L3 Case Study

This section introduces a case study that demonstrates how to apply the proposed methodology. The case study, from the railway domain, is an excerpt of the ETCS-L3 trackside-train communication and control mechanisms presented in Subsection 2.2.

The System Model. Let us consider two on-board macro-functionalities of ETCS-L3:

- the braking supervision, which controls the speed of the train and, in case of emergency, activates the train brakes;
- the movement supervision, which controls the position of the train, on the base of what is received from the track-side and on the actual position, commands an emergency brake.

The system structural view, for these two macro-functionalities, is given in Fig. 5 by an IBD and in Fig. 6 by a Class Diagram (CD). In addition, Fig. 7 reports details of the blocks, presenting relevant properties and operations.

The BD in Fig. 5 presents four main components: *Trackside*, *Train*, *Brake* and *Odometer*. The latter is responsible for providing exact kinematic information—i.e. speed and position—to the other components. Internal to the Train component, we identify the *Braking Supervision* and the *Movement Supervision*. The BD also reports the ports, signals and exchanged data on the item flows, used by the components in their interactions. Some of these interactions include the following:

- Braking Supervision communicates braking intentions to Brake via the *command* signal (from *cmd* port to *getCmd* port), the flow conveys the *BrakeCommand* type;
- Movement Supervision asks for an emergency stop to Braking Supervision via the *emergencyStop* signal (from *sendBrake* port to *recvES* port), the flow conveys the *EmergencyStop* type;
- Odometer sends two float values respectively to:
  - Movement Supervision, this is the case of the position (from *updtPosition* to *getPosition* via the *updatedPosition* signal);
  - Braking Supervision, this is the case of the speed (from *updtSpeed* to *getSpeed* via the *updatedSpeed* signal).

Figures 8 and 9 depict initial SMDs for both, the Movement Supervision and Braking Supervision, offering the Behavioural view. To complete this initial model, four ETCS-L3 requirements make up the Requirements view, see Listings 6, 7, 8 and 9. These requirements are expressed using the Given–When–Then Gherkin syntax.

Table 4 Requirements and patterns

Requirement	Matching MetaReq
REQ-1	<G1,W1,T1 and T2>
REQ-2	<G2,W2,T3>
REQ-3	<G1,W3,T3 and T2>
REQ-4	<G2,W4,T5>

Listing 6 ETCS-L3 requirements: REQ-1

```
Given a Braking Supervision in running,
    When the Movement Supervision
   sends an Emergency Stop, Then the
   Braking Supervision brakes and goes
    in braking.
```

Listing 7 ETCS-L3 requirements: REQ-2

```
Given a Braking Supervision, When the
   Braking Supervision brakes, Then
   the Braking Supervision sends to
   Brake a command with active
   BrakeCommand.
```

Listing 8 ETCS-L3 requirements: REQ-3

```
Given a Braking Supervision in braking,
    When the speed of the Train is 0,
   Then the Braking Supervision sends
   to Brake a command with NonActive
   BrakeCommand and goes in running.
```

Listing 9 ETCS-L3 requirements: REQ-4

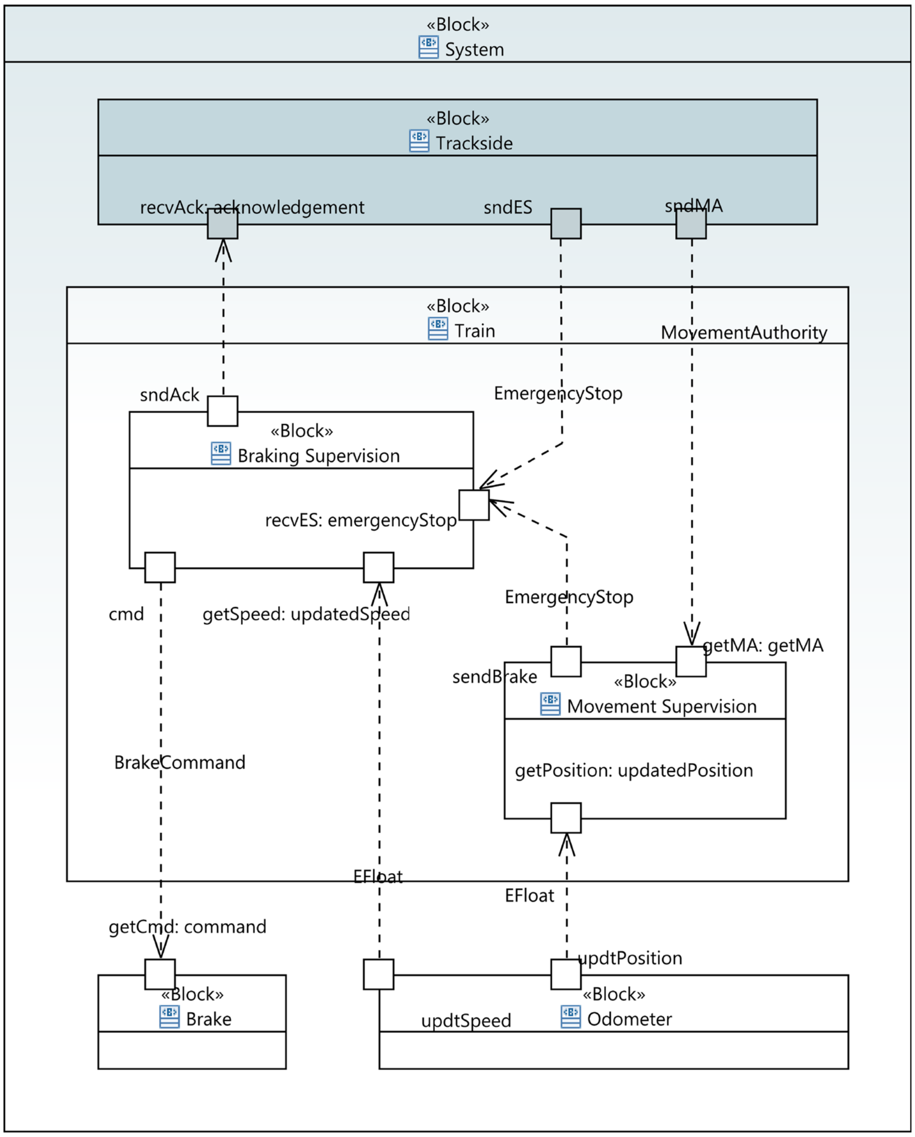
```
Given a Movement Supervision, When the
   Movement Supervision receives an
   updated position with a position,
   Then Movement Supervision sets the
   position of the Train to the
   position value of updated position.
```

Starting from this initial model, the model element names are extracted to make the concrete grammar and enable the matching of the requirements:

- Blocks: Train, Braking supervision, Brake, Movement Supervision, Odometer and Trackside.
- States: running, braking, updateWaiting and waiting.
- Signals: Emergency Stop Message and Acknowledge.
- Operations: activates, deactivates and brake.
- Properties: speed, position and safetyDistance.

Matching the requirements. Starting from the four requirements, Table 4 reports the meta-requirements matched by the Tool Layer.

Fig. 5 System blocks



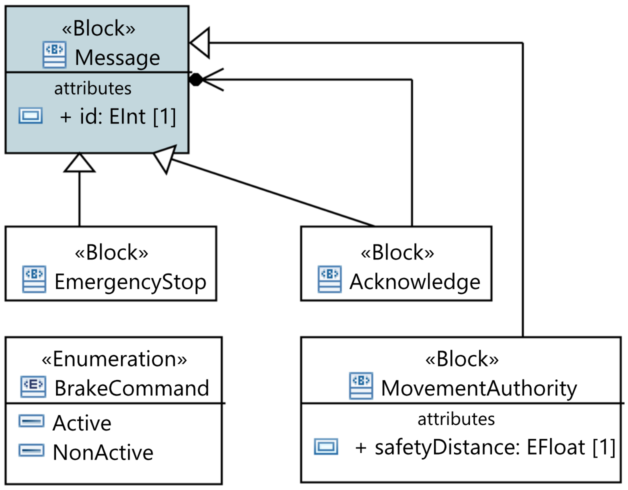


Fig. 6 Exchanged data

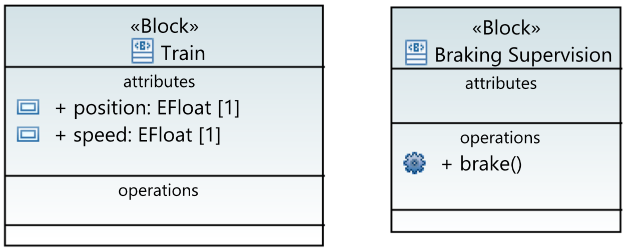


Fig. 7 Block property

Model Completion. Given all the working hypotheses, it is now possible to apply the second part of the workflow in Fig. 3, as described in Sect. 4. Hence, analysing and translating each requirement. Figures 10 and 11 report the refined SMDs of the Braking Supervision and the Movement Supervision components, respectively.

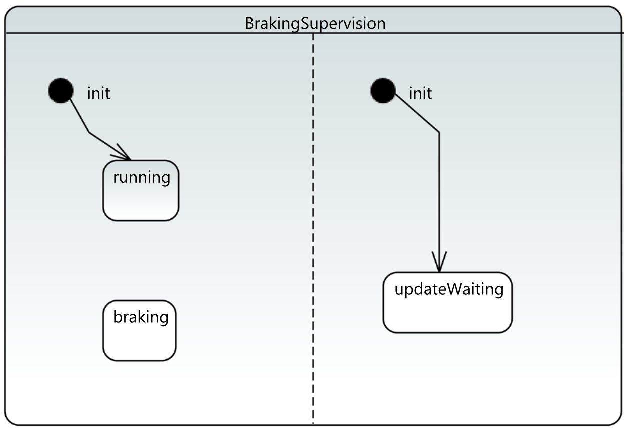


Fig. 8 Braking supervision: SMD

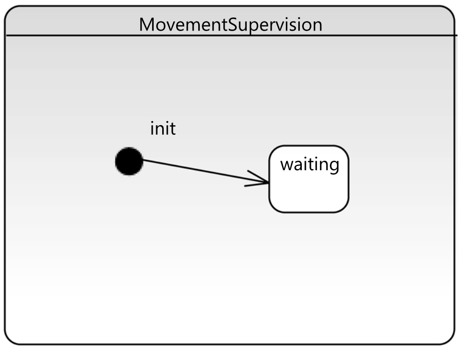


Fig. 9 Movement supervision: SMD

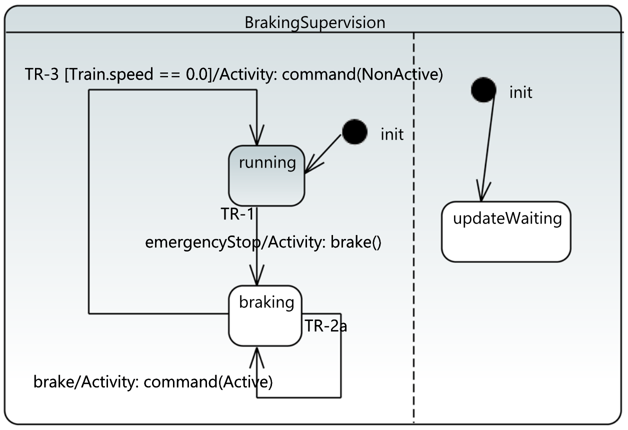

BrakingSupervision
TR-3 [Train.speed == 0.0]/Activity: command(NonActive)
init
init
running
updateWaiting
TR-1
emergencyStop/Activity: brake()
braking
TR-2a
brake/Activity: command(Active)

Fig. 10 Braking supervision: refined SMD

In the SMD of Fig. 10, the following transitions have been created:

– **TR-1**: it starts from the *running* state (G1), ends in the *braking* state (T2), is triggered by *emergencyStop* (W1) and activates the *brake()* operation (T1);
– **TR-2a**: self-transition on the *braking* state (G2), triggered by the activation of the *brake()* operation (W2), and sending a braking *command* to the *Brake* with the *Active* parameter value (T3). It is important to underline that other two transitions are generated (T2b, from/to the *running* state, and T2c, from/to the *updateWaiting* state); these transitions are not reported in the diagram for simplicity.
– **TR-3**: it starts from the *braking* state (G1) and ends in the *running* state (T2), is guarded by the condition `Train.speed == 0.0` and, when executed, sends a message via the command signal (T3);

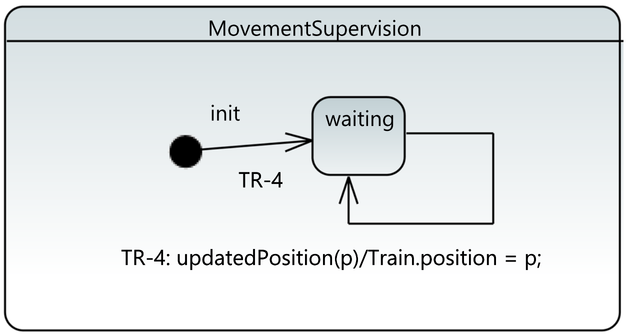


Fig. 11 Movement supervision: refined SMD

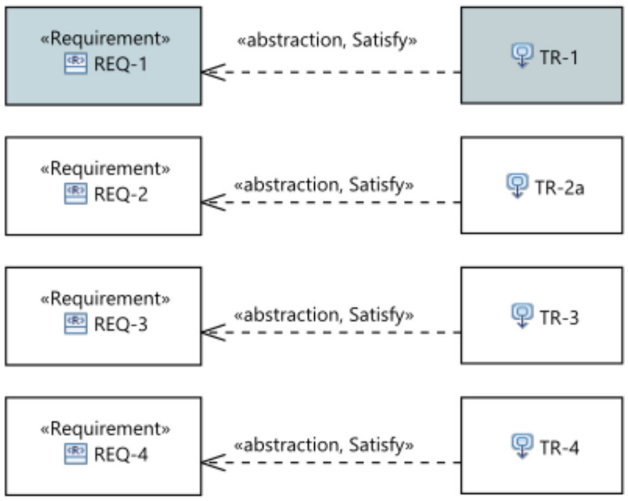


Fig. 12 RD of the case study

The SMD of Fig. 11 has been enriched with the self-transition **TR-4** that concerns the *waiting* state of the Movement Supervision component (G2), and it is triggered by the reception of the *updatePosition* signal with value *p* (W4) and assigns such a value to the *position* property of the *Train* block (T5).

To complete the application, the RD showing the <<satisfy>> relationships between generated transitions and system requirements is reported in Fig. 12.

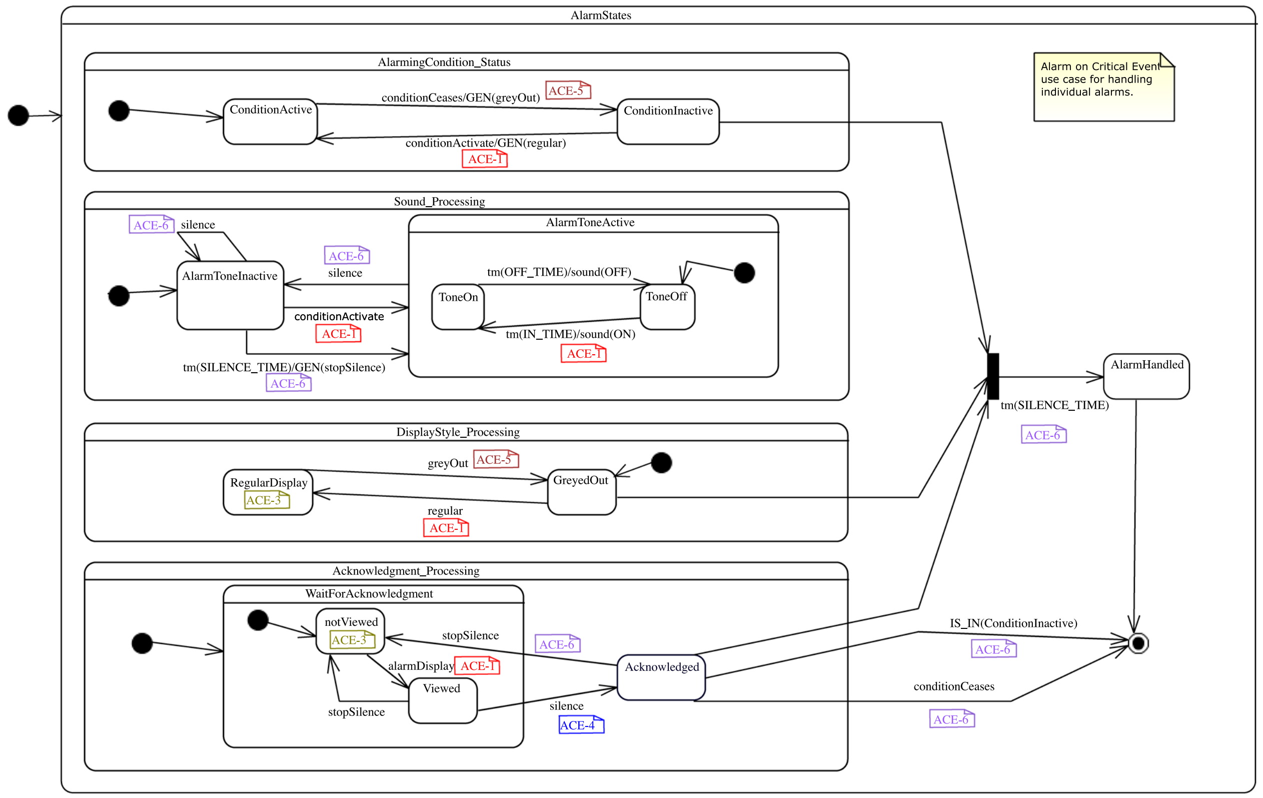


Fig. 13 UML state machine from [18] with traced requirements annotated

11 The Alarm on Critical Events Case Study

This case study leverages a critical event-based system, in the medical domain, taken from the literature [18]. The objective of this section is to demonstrate the effectiveness of the approach. We do this by (qualitatively) measuring how close the models obtained by the proposed methodology are to the model presented in the literature. The following steps are taken:

1. Application of the methodology to the case study, and
2. discussion of the results.

11.1 Application of the methodology

The case study is an alarming system that visually and audibly alerts the attending anaesthesiologist when the patient is at imminent risk, so to take appropriate action. In [18], behavioural requirements (Table 5) are introduced and partially mapped to a UML state machine (Fig. 13). The aim in [18] is to demonstrate the suitability of the UML state machines in capturing and analysing non-trivial cause–effect relationships in reactive systems. In particular, the UML state machine includes annotations "ACE-n", which trace the requirements in Table 5. These annotations have been defined by interpreting the informal mapping provided in [18]. Observe that all the requirements, but "ACE-2"—which addresses multiple alarms—are considered in the UML state machine. Also, all the requirements, but "ACE-3"—which states a negation of an event occurrence—express cause–effect relations and are mapped to transitions in the state machine.

Requirements view. To apply our approach, we had to pre-process the original requirements: 1) rephrase them as Given–When–Then statements and 2) refine them by taken contextual information into account.

Specifically, the rephrasing was carried out with the help of ChatGPT, where the following *prompt* was posed:

> Write the following requirement in Given–When–Then form: "original requirement (see Table 5)"

The resulting GWT requirements were manually refined considering the contextual information in the SysML model. Table 6 shows the results of the pre-processing for the "ACE-1" requirement presented in Table 5. "ACE-1", in Table 6, appears as rephrased by ChatGPT, and "REQ−1.1-REQ−1.4" represent its contextualized requirements. Appendix A contains an exhaustive description of the pre-processing results for the remaining requirements.

Table 5 Behavioural requirements from [18]

Req. ID	Description
ACE-1	When an alarming condition occurs, it shall be annunciated—that is, a meaningful alarm message (including the time of occurrence, the type of alarm, the source of the alarm and the likely cause of the alarm) shall be displayed and an alarming tone shall be sounded
ACE-2	When multiple alarms are being annunciated, they shall be displayed in order of severity first, then in order of occurrence, newest first
ACE-3	If an annunciated alarm isn't displayed (because higher criticality alarms are being displayed and there is insufficient space to display the alarm in question), then it cannot be acknowledged without first being displayed on the screen
ACE-4	Alarms must be explicitly acknowledged by the user pressing the Alarm Ack button after they have occurred even if the originating alarming condition has been corrected
ACE-5	If the originating condition of an alarm has been corrected but the alarm has not yet been acknowledged, then the display of the alarm message shall be greyed out. All other alarm messages shall be displayed in the normal colour
ACE-6	The Alarm Ack button shall cause the audible alarm sound to be silenced but does not affect the visual display of the alarm message. The silence shall hold for 2 min. If the alarm condition ceases after the acknowledgement but before the silence period times out, then the alarm shall be dismissed. If, after the silence period has elapsed, the originating condition is still valid or if it has reasserted itself during the silence period, then the alarm shall be reannunciated

Table 6 Pre-processing ACE-1: Alarming Condition Occurrence

Step: ChatGPT		
ACE-1'		Given an alarming condition, when the condition occurs, then it shall be annunciated by displaying a meaningful alarm message (including the time of occurrence, the type of alarm, the source of the alarm and the likely cause of the alarm) and sounding an alarming tone.
Step: Refinement		
REQ ID	Contextual information	Description
REQ−1.1	AlarmingCondition, ConditionInactive, conditionActivate, regular, DisplayStyle, ConditionActive	Given an AlarmingCondition in ConditionInactive, when the AlarmingCondition receives a conditionActivate, then the AlarmingCondition sends a regular event to DisplayStyle and the AlarmingCondition goes into ConditionActive
REQ−1.2	DisplayStyle, GreyedOut, regular, alarmDisplay, Acknowledgement, RegularDisplay	Given a DisplayStyle in GreyedOut, when the DisplayStyle receives a regular event, then the DisplayStyle sends an alarmDisplay to Acknowledgement and the alarmDisplay goes into RegularDisplay
REQ−1.3	Sound, AlarmToneInactive, conditionActivate, setAlarmTone(ON), AlarmToneActiveToneOn	Given a Sound in AlarmToneInactive, when the Sound receives a conditionActivate, then the Sound calls setAlarmTone with ON and the Sound goes into AlarmToneActiveToneOn
REQ−1.4	Acknowledgement, WaitForAcknowledgementNotViewed, alarmDisplay, WaitForAcknowledgementViewed	Given an Acknowledgement in WaitForAcknowledgementNotViewed, when the Acknowledgement receives an alarmDisplay, then the Acknowledgement goes into WaitForAcknowledgementViewed

The automation of this pre-processing step goes beyond the scope of this work, but certainly it could be improved by applying *prompt* engineering techniques.

Structural view. The structural view has been defined from scratch, as it is not developed in [18]. The system is made of four main components: AlarmingCondition, Sound, Display Style and Acknowledgement, as modelled in Fig. 14. Note

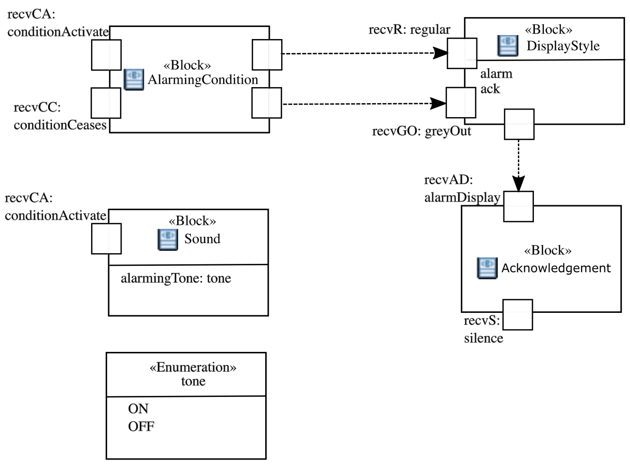


Fig. 14 Structural view: block diagram

that with the aim of eventually facilitating the comparison of the results, these components have been named using the (first part of the) names of the parallel regions in Fig. 13—e.g. AlarmingCondition component from AlarmingCondition _Status region. The components interact through signal sending and reception: signal types correspond to the triggers in Fig. 13—e.g. *conditionActivate* triggers in the AlarmingCondition_Processing and Sound_Processing concurrent regions.

Behavioural view. The initial behavioural view consists of four state machines, each one representing the behaviour of a component in Fig. 14. They only include initial and simple states—see Fig. 16 in Appendix A. The names of the model elements of the state machines are similar to those of the original model in Fig. 13. On the other hand, since our approach does not currently support composite states, we have considered nested states to keep trace of the original composite state. The name of a nested state is obtained by concatenating the name of the parent state with the name of the nested state—e.g. AlarmToneActive concatenated with ToneOn then resulting in AlarmToneActiveToneOn.

Matching the requirements. Once obtained the requirements, the meta-requirements identification phase begins. In particular, the objective is to identify those clause templates, listed in Table 1, that make up the requirements. As mentioned in Sect. 7, the clause templates work as a relation between the behavioural model's components and the requirements' contextual information.

Then, to perform the identification, the workflow, shown in Fig. 3, is applied to each requirement. The result is the mapping of each requirement with a meta-requirement and the extrapolation of roles as shown in Tables 12-17.

Considering Table 12, REQ−1.1 matches with meta-requirement <`G1,W1,T7 and T2`> and it provides information regarding the behaviour and change of state of the *AlarmingCondition* when it receives the *conditionActive* signal. REQ−1.2 matches the same meta-requirement as REQ−1.1, but it expresses the transition from the starting state *GreyedOut* to the final state *RegularDisplay* of the *DisplayStyle* component when the *regular* signal is received.

The meta-requirement identified for REQ−1.3 is different, and in fact it groups the clause templates <`G1,W1,T1 and T2`>. In this requirement, the *Sound* component undergoes a change of state after receiving the *conditionActive* signal, and in addition, the *setAlarmTone* method is called. Finally, REQ−1.4 matches with <`G1, W1,T2`>, then describing the behaviour of the *Acknowledgement* component.

Model Completion. Once the matching of REQ-1.x requirements is completed, a model fragment is created from them for each state machine. These new model elements are added to the initial SysML model, augmenting it.

The creation of the transitions is obtained using the information outlined by the meta-requirements and applying the mapping rules collected in Table 2.

The first state machine depicted in Fig. 16 is the initial state machine of the *AlarmCondition* block. The states to be connected by a transition are *ConditionActive* and *ConditionInactive*. REQ−1.1 comes into play here (see Table 12) since the contextual block is precisely *AlarmingCondition*. Moreover, the clause template `G1` suggests as initial state *ConditionInactive* and `T2` suggests as final state *ConditionActive*. Consequently, the transition is created following the order of the states and enriched by the information of the clause templates `W1` and `T7`. The trigger is the signal *conditionActive*, and the action is the event regular (affecting *DisplayStyle* block). The result is the first state machine depicted in Fig. 15.

The second state machine, of Fig. 16, represents the behaviour of *Sound_Processing*. The initial states are *AlarmToneInactive*, *AlarmToneActiveToneOff* and *AlarmToneActiveToneOn*. The requirement for expressing the correct behaviour of the Sound component is REQ−1.3. Proceeding in order and considering Table 12, the meta-requirement suggests that one behaviour of *Sound_Processing* is the change of states between *AlarmToneInactive* and *AlarmToneActiveToneOn* (clause templates `G1` and `T2`). As a result, the new transition connects these two states. Moreover, it is initiated by the signal *conditionActive* and calls the function *setalarmTone()* (clause templates `W1` and `T1`). Once the meta-requirements are translated, the complete and detailed state machine, shown in Fig. 15, is obtained.

The *DisplayStyle* component is the subject of the next state machine to be completed. Here the relationship between the *RegularDisplay* state and the *GreyedOut* state needs to be defined employing the meta-requirement of REQ−1.2. The *GreyedOut* is, indeed, detected as the source state of the translation (`G1`), and `T2` indicates *RegularDisplay* as the target state. Moreover, the clause template `W1` indicates that

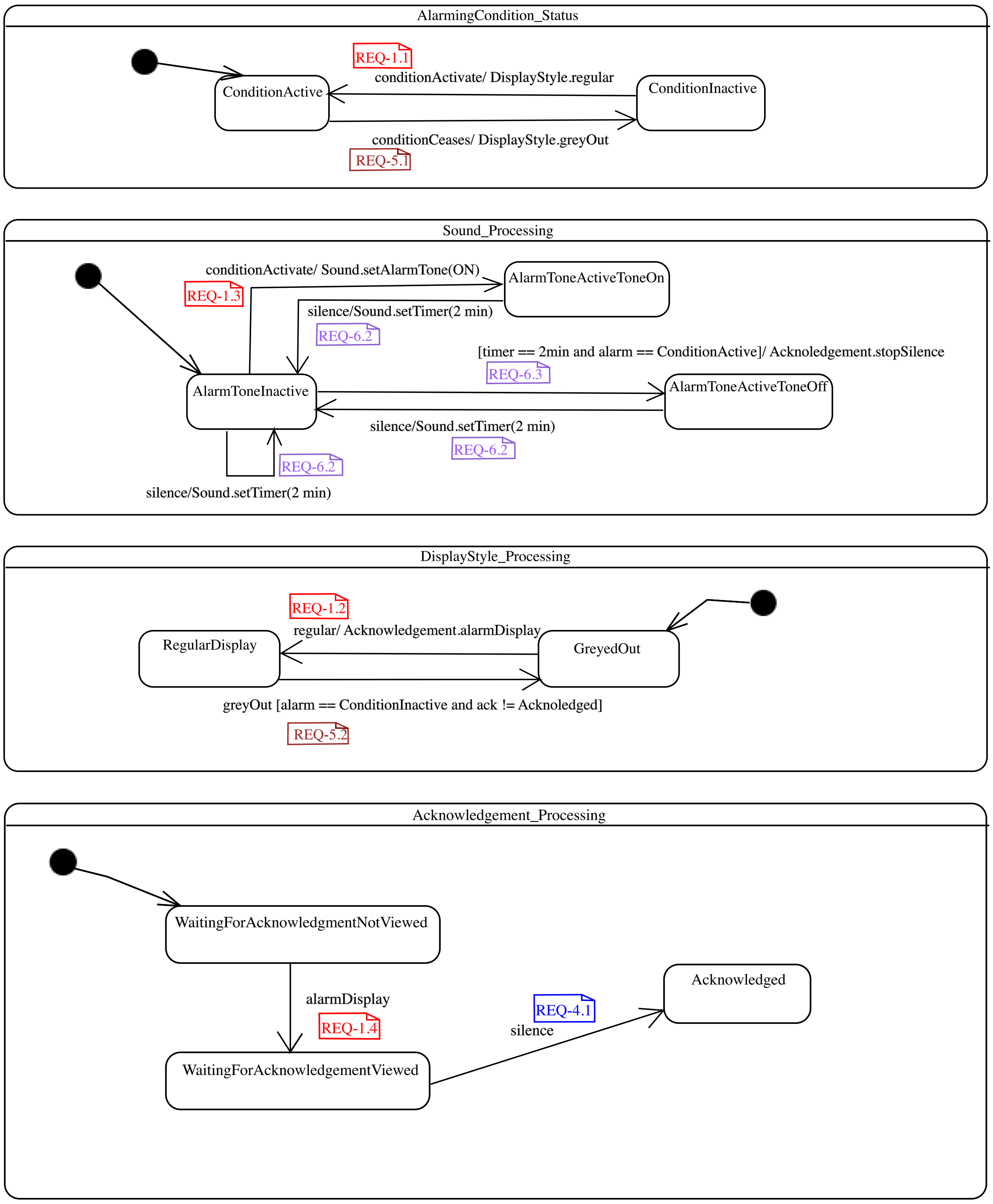

AlarmingCondition_Status
REQ-1.1
conditionActivate/ DisplayStyle.regular
ConditionActive
ConditionInactive
conditionCeases/ DisplayStyle.greyOut
REQ-5.1
Sound_Processing
conditionActivate/ Sound.setAlarmTone(ON)
REQ-1.3
AlarmToneActiveToneOn
silence/Sound.setTimer(2 min)
REQ-6.2
[timer == 2min and alarm == ConditionActive]/ Acknoledgement.stopSilence
REQ-6.3
AlarmToneInactive
AlarmToneActiveToneOff
silence/Sound.setTimer(2 min)
REQ-6.2
REQ-6.2
silence/Sound.setTimer(2 min)
DisplayStyle_Processing
REQ-1.2
regular/ Acknowledgement.alarmDisplay
RegularDisplay
GreyedOut
greyOut [alarm == ConditionInactive and ack != Acknoledged]
REQ-5.2
Acknowledgement_Processing
WaitingForAcknowledgmentNotViewed
alarmDisplay
REQ-1.4
Acknowledged
REQ-4.1
silence
WaitingForAcknowledgementViewed

Fig. 15 Behavioural view: state machines of AlarmingCondition, Sound, DisplayStyle and Acknowledgement, respectively, after the application of the approach

the transition occurs as it is generated by the reception of the *regular* signal and, in turn, triggers another signal affecting the *Acknowledgement* block (clause template `T7`).

The *Acknowledgement* block in fact shifts from state *WaitForAcknowledgementNotViewed* to *WaitForAcknowledgement Viewed* according to REQ$-$1.4, in particular according to `G1` and T2. Finally, W1 completes the creation of the SysML transition as in Fig. 15.

11.2 Discussion of the case study results

We have evaluated the effectiveness of the approach by measuring how close the resulting state machines are to the state machine presented in the literature [18]. Specifically, we compare the added transitions with the model elements (transitions and states) of the original model.

Quantitatively, 12 out of 15 transitions and the two states of the original model (Fig. 13) have a counterpart in the created model, which is a promising result of the automatic generation capability of the proposed approach[9].

From a qualitative point of view, we carried out a fine grained analysis with the aim of evaluating the semantic equivalence between the model elements of the two models. In particular, the analysis relies in Table 18, which details the mapping between the elements of the two models. The detailed results of such analysis are reported in Appendix (subsection A.4).

As a global assessment, we can conclude that the semantic equivalence holds for a few transitions (rows 1, 2, 11 and 14 of Table 18), whereas the remaining transitions in the derived model are even more informative than the mapped model elements in the original model. For example, effect or guard elements are added in the transitions of the derived model, while there were not in the corresponding transitions of the original model (e.g. rows 5, 6, 8–10 of Table 18). This result is not surprising, since the pre-processing of the original requirements entails the inclusion of contextual information in the derived requirements, resulting in more precise specifications.

12 Discussion and Conclusions

This paper develops an approach to semi-automatically complete SysML models, on the basis of system-level requirements expressed using the Given–When–Then paradigm. Hence, the paper shows, using a running example and two case studies from different application domains, how to complete a SysML model by adding transitions in an SMD between pre-existing states.

The final impact, that the semi-automation of the approach can achieve, is the creation of a Computer Aided Modelling (CAM)-like tool that supports System Analysts. Also, the work aims to propose a technically sound approach able to be integrated into current industrial processes. Notwithstanding these aims, the paper opens questions, not fully addressed here, that deserve a brief discussion, also to understand future areas of investigation.

The most important question, for us, is related to the role of the *Parsing* phase of the workflow. Our text-based requirement specification relies on the definition of a formal EBNF-based grammar. However, real-world industrial practice should keep a balance between flexibility and the existence of internal specifications and/or design standards. To this aim, a balance between a grammar and the usage of NLP techniques is needed. In future work, we will research a way of harnessing these two needs.

To improve the flexibility of the parsing phase, some techniques could be used in cooperation with the formal parsing one. Two examples are the usage of lexical and syntactical tools proposed in NLP—e.g. lemmatization, Part-of-Speech (POS) tagging—and/or the application of probabilistic parsing techniques to boost the capability of compilers to recognize patterns [55].

Another important open point is the need to transform requirements into the BDD paradigm. Even if BDD is widespread and has demonstrated its easy and effective application in industrial settings—like emphasized in Sect. 2.1—we know that, in some legacy systems, the requirements should be rewritten from scratch, then limiting the applicability of the approach. To this aim, generative technologies can help. The Alarm on Critical Event system case study has shown the capabilities of current transformers for this task. The non-BDD original requirements have been transformed using the well-known OpenAI's Generative Pre-trained Transformer (GPT−3.5),[10] by means of a given *prompt*. Furthermore, *prompt* engineering techniques, when fine-tuned with system contextual information, help to enrich the requirements, hence improving the two pre-processing stages. In our view, this approach could lead to a more precise and efficient automation of the methodology. Consequently, the Alarm on Critical Event case study illustrates a practical and effective application of requirements pre-processing using a Large Language Model.

There are several scientific papers addressing the consistency and completeness of requirements. They span from checking the consistency—at both, textual level (e.g. in [48]) and involved model (e.g. in [26])—to verify completeness aspects, as in [20] or in [3], where SysML model is involved. Even if this paper does not explicitly address these two features, the proposed approach enables the verification of both properties. As an example, the verification of completeness could be addressed by constructing some SysML model queries that retrieve "model anomalies" as isolated states, transitions that are not mapped on any requirements or requirements that are not satisfied by any model element. On

[9] Observe that the three transitions—annotated with *ACE-6* in Fig. 13—which have the join node (or the final state) as source (or target), have no counterpart in the new model, as the join node and the final states were not considered in the initial behavioural view.

[10] https://chat.openai.com/

the other hand, consistency of requirements could be checked using Gherkin, by means of static semantics checkers on the constructed ASTs. As a further consideration, the approach should rely on such tools in this phase since in the development processes of critical systems the usage of “AI-based” approaches is hard to certificate. This is also another motivation to the semi-automatic nature of the approach itself.

A third point of discussion is related to the possibility of non-optimal model refinement in the case of complex requirements. If considering a G2 given clause, the approach prescribes the generation of a transition for each state of the component. However, this could be optimized, for example, by considering hierarchical states. As in classical compilers, optimization techniques could be explored to improve the readability of the improved model.

Finally, full automation and its impact on the end user must be taken into account. Our main goal is not to build an interactive tool that refines the model, while the user carries out the modelling activity, but to develop a back-end tool, that is able to transform a SysML model into another model. To develop such a tool, developers would rely on existing technologies, spanning from parser generators (e.g. SableCC), to template engines (e.g. Te4j), to SysML model query and manipulation libraries (e.g. the ones in the Eclipse Modelling Framework (EMF) ecosystem). For the Knowledge layer, on the other hand, a document-oriented database as MongoDB is suggested. Papyrus[11] is an open-source tool that serves a valid front-end to build the User layer. As it is based on the Eclipse platform, the entire approach can be implemented as an Eclipse plugin. Further, investigations could be oriented to develop a sort of co-pilot, which improves productivity and user experience.

Summarizing, future work will be oriented to:

- the completion of Tool Layer prototyping and tuning of the proposed approach;
- injection of NL-based techniques into the Parser component;
- study of the possible optimization on refined SysML model;
- training of Large Language Models (LLMs) to transform free-expressed requirements to the BDD paradigm.

Another long-term research will also be oriented to reducing as much as possible the pre-existing, manually made model part needed to enable the approach. As an example, the first step could be constituted by inferring automatically the states and adding them to the SysML model. Then, also class/block diagrams could be inferred according to some existing scientific papers (e.g. [46, 54]).

[11] https://eclipse.dev/papyrus/

Acknowledgements The work of Maria Stella de Biase is granted by PON Ricerca e Innovazione 2014/2020 MUR—Ministero dell’Università e della Ricerca (Italy)—with the PhD programme XXXVI cycle. This research has been supported by the Shift2Rail Joint Undertaking under the European Union’s Horizon 2020 research and innovation programme under Grant agreement no. 101015416 PERFORMINGRAIL. S. Bernardi and J. Merseguer were supported by the Spanish Ministry of Science and Innovation (Damocles project, ref. PID2020-113969RB-I00) and the Aragonese Government (Disco research group, ref. T21-23R).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

A Alarm on Critical Event system (detailed)

This appendix reports the intermediate results of the pre-processing steps (subsection A.1), the initial SysML behavioural view (subsection A.2), the mapping of refined GWT requirements to meta-requirements (subsection A.3) and the mapping of the model elements in the original model to the transitions in the derived model (subsection A.4).

A.1 Pre-processing of original requirements

The results of the pre-processing are shown in Tables 6, 7-11. Each table details:

- The rephrasing of each requirement—labelled ACE-n’, where ACE-n is the original requirement in Table 5, and
- the refinement of the rephrased requirement that considers the contextual information in the structural and behavioural views—Figs. 14 and 16, respectively.

Observe that a rephrased requirement ACE-n’ can be refined in a set of m contextualized requirements, which are labelled REQ-n.1 …REQ-n.m.

ACE-2 The requirement ACE-2 concerns multiple alarms occurrence, and it was not considered in the original state machine of [18] (see Fig. 13). For completeness, we also consider this requirement and Table 7 details the corresponding

Table 7 Pre-processing ACE-2: Multiple Alarms Occurrence

Step: ChatGPT		
ACE-2’		Given multiple alarms are being annunciated, when displaying the alarms, then they shall be displayed in the following order: First by severity (from highest to lowest), then by occurrence, with the newest alarms displayed first.
Step: Refinement		
REQ ID	Contextual information	Description
REQ−2.1	DisplayStyle, GreyedOut, regular, numberOfAlarms, ordering	Given a DisplayStyle in GreyedOut, when DisplayStyle receives a regular event and the number of alarms of DisplayStyle is greater than zero, then the DisplayStyle orders the alarms by severity first, then in order of occurrence, newest first

Table 8 Pre-processing ACE-3: Not Displayed Alarm

Step: ChatGPT		
ACE-3’		Given an annunciated alarm is displayed (because there is sufficient space to display the alarm in question), when attempting to acknowledge the alarm, then it can be acknowledged.
Step: Refinement		
REQ ID	Contextual information	Description
REQ−3.1	DisplayStyle, GreyedOut, regular, sufficientSpace, higherCriticalityAlarms, alarmDisplay, Acknowledgement, RegularDisplay	Given a DisplayStyle in GreyedOut, when the DisplayStyle receives a regular event and the sufficient space of DisplayStyle is true, then the DisplayStyle sends an alarmDisplay to Acknowledgement and the DisplayStyle goes into RegularDisplay
REQ−3.2	Acknowledgement, WaitForAcknowledgementViewed, silence, Acknowledged	Given an Acknowledgement in WaitForAcknowledgementViewed, when the Acknowledgement receives a silence, then the Acknowledgement goes in Acknowledged

Table 9 Pre-processing ACE-4: Explicit Acknowledgement Alarm

Step: ChatGPT		
ACE-4’		Given an alarm has occurred, when the user presses the Alarm Ack button, then the alarm must be explicitly acknowledged, even if the originating alarming condition has been corrected.
Step: Refinement		
REQ ID	Contextual information	Description
REQ−4.1	silence, Acknowledgement	Given an Alarm Ack button in alarm occurred, when the user presses the Alarm Ack button, then the Alarm Ack button sends a silence to Acknowledgement
REQ−4.2	Acknowledgement, WaitForAcknowledgementViewed, silence, Acknowledged	Given an Acknowledgement in WaitForAcknowledgementViewed, when the Acknowledgement receives a silence, then the Acknowledgement goes into Acknowledged

refined requirement REQ−2.1. Nevertheless, the application of the proposed approach considering also REQ−2.1 produces inconsistencies in the state machine of DisplayStyle component which will be discussed in subsection A.3.

ACE-3 The requirement ACE-3 is stated as a negation of an event—i.e. an alarm is not displayed—and, despite the rest of the requirements in Table 5, it was mapped onto states of the involved regions (see Fig. 13, state “RegularDisplay" of DisplayStyle_Processing and state "NotViewed" of Acknowledgement_Processing). Since our approach enables to enrich state machines with transitions, which represent cause–effect relations, we have considered the negation of ACE-3 as follows:

> If an annunciated alarm is displayed (because there is sufficient space to display the alarm in question), then it can be acknowledged.

Table 8 shows the rephrased and refined requirements of such negation.

ACE-4 The requirement ACE-4 states the explicit acknowledgement of an alarm occurrence by the user through "Alarm Ack button" pressing, and it has been partially mapped in the original state machine (see Fig. 13, transition from "Viewed" to "Acknowledged" of the Acknowledgement_Processing region). Indeed, the original state machine does not include the behaviour of the user interface. Although we fully refined this requirement for completeness purposes (see Table 9), observe that REQ−4.1 cannot be mapped by applying the proposed approach since contextual information is missing in the structural and behavioural views of Figs. 14 and 16, respectively (e.g. Alarm Ack button). Therefore, only REQ−4.2—which is the same as REQ−3.2—has been mapped by applying the proposed approach.

ACE-5 The requirement ACE-5 deals with the ceasing of an alarm condition and its refinement leads to two requirements REQ−5.1 and REQ−5.2 (see Table 10). In particular, REQ−5.1 states the occurrence of the alarm correction and, therefore the ceasing of the alarm condition (the "given" part of ACE-5'). REQ−5.2 involves the DisplayStyle component, which has the visibility on the states of AlarmCondition and Acknowledgement components (see Fig. 14, attributes "alarm" and "ack" of the DisplayStyle block).

ACE-6 The requirement ACE-6 is the most complex: it concerns silencing an audible alarm for a time period, and it is partially refined as shown in Table 11. Indeed the ACE-6' part "if the alarm condition ceases after the ack but before the silence period time-out then the alarm shall be dismissed" has not been refined since it involves global states/node in the original state machine (Fig. 13) which we have not considered in the initial behavioural view (Fig. 16).

A.2 Initial SysML behavioural view

Figure 16 shows the initial behavioural view, where a state machine is associated with each component. The state machines only include simple states, since the proposed approach does not currently deal with composite states.

A.3 Mapping refined requirements to meta-fragments

Refined requirements REQ-n.m have been matched to requirement patterns. Tables 12, 13, 14, 15, 16, and 17 present such matching.

In particular, each table details requirements REQ-n.1-REQ-n.x corresponding to the original requirement ACE-n. For each refined requirement, the table reports the detected requirement pattern (**MetaReq**) with the matching of the contextual information in the initial model to the roles in the pattern (**Role matching**). All the refined requirements, but REQ−4.1, have been fully matched. REQ−4.1 is only partially matched due to the absence of contextual information related to the user interface in the initial SysML model (see Table 15).

The translation of all the detected requirement patterns but the REQ−2.1 leads to the resulting state machines of Fig. 15, where the tracing of the added model elements (transitions, trigger, effects, guards) to the refined requirements is shown as annotations. In particular, there is a one-to-one mapping between the refined requirements and transitions for almost all the requirements. The exceptions are the following:

- REQ−6.2, where the translation produces multiple transitions; and
- REQ−1.2, REQ−3.1, REQ−3.2, REQ−4.1 and REQ−6.1, where the translations produce refinements of transitions.

Creation of multiple transitions. REQ−6.2 matches the meta-requirement `<G1,W1,T2 and T4>` (Table 17) which is translated to three transitions, one for each simple state of Sound_Processing state machine according to the meta-fragment associated with the `G1` clause in Table 2. All the transitions have the same trigger, effect and target state.

Refinement of transitions. Let us consider requirements REQ−1.2 and REQ−3.1 first, which concern the DisplayStyle_Processing state machine. They match the meta-requirements `<G1,W1,T7 and T2>` (Table 12) and `<G1,W1 and W3, T7 and T2>` (Table 14), respectively, where the second is a refinement of the first one. Therefore, the translation of the second meta-requirement refines the already created transition from GreyedOut to RegularState by adding the guard.

Requirements REQ−3.2, REQ−4.1 and REQ−6.1 concern the Acknowledgement_Processing state machine. In particular, REQ−3.2 and REQ−4.1 are identical and they match the meta-requirement `<G1,W1,T2>` (Tables 14 and 15). Therefore, the transition from WaitForAcknowledgementViewed to Acknowledged with the trigger event (silence) is created after the first occurrence of the meta-requirement detection. The second occurrence of meta-

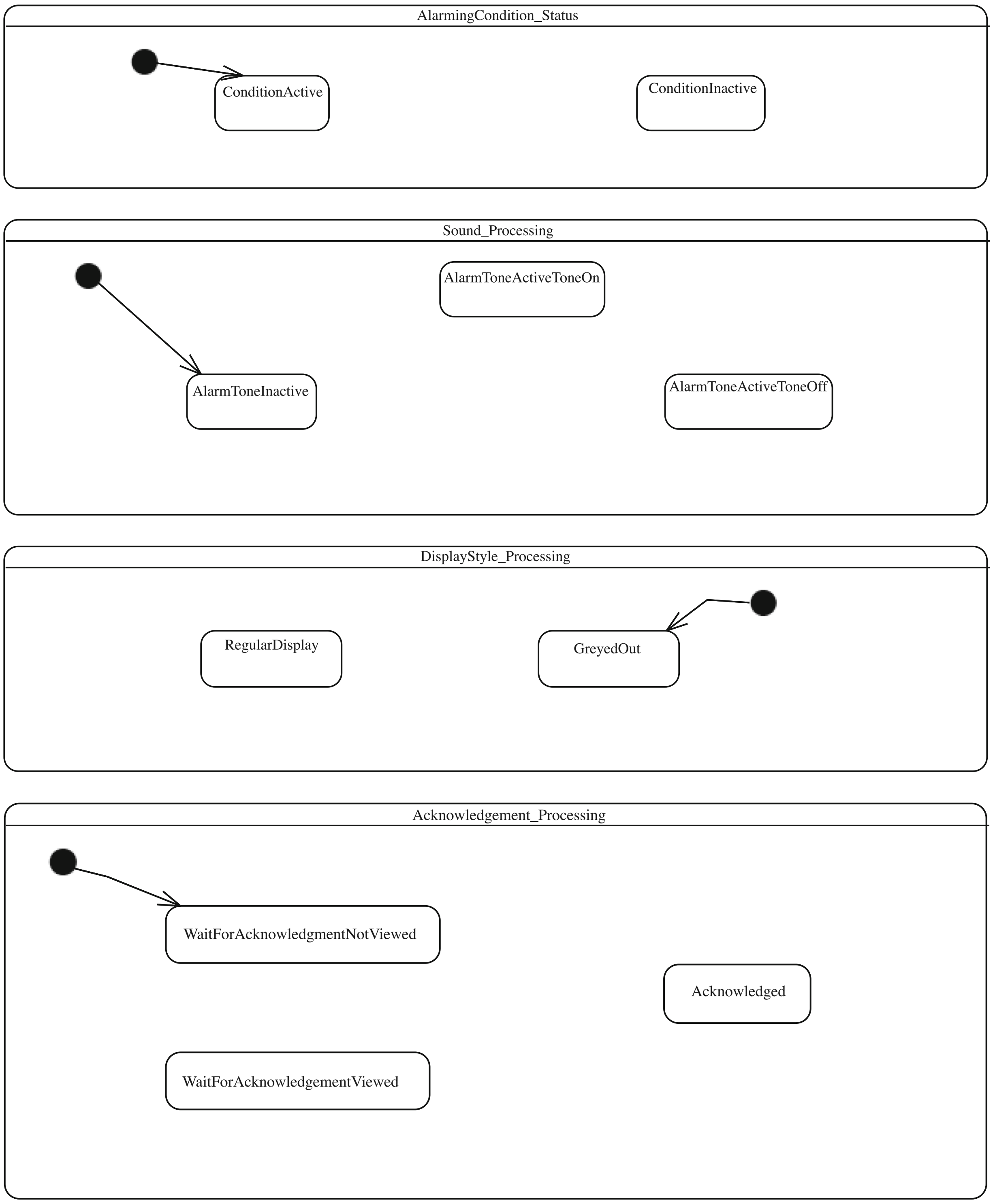


Fig. 16 Behavioural view: initial state machines of AlarmingCondition, Sound, DisplayStyle and Acknowledgement, respectively

Table 10 Pre-processing ACE-5: Alarm Condition Ceases

Step: ChatGPT		
ACE-5’		Given the originating condition of an alarm has been corrected, when the alarm has not yet been acknowledged, then the display of the alarm message shall be greyed out, and all other alarm messages shall be displayed in the normal colour.
Step: Refinement		
REQ ID	**Contextual information**	**Description**
REQ−5.1	AlarmingCondition, ConditionActive, conditionCeases, DisplayStyle, greyOut, ConditionInactive	Given an AlarmingCondition in ConditionActive, when the AlarmingCondition receives a conditionCeases, then the AlarmingCondition sends a greyOut to DisplayStyle and the AlarmingCondition goes into ConditionInactive
REQ−5.2	DisplayStyle, RegularDisplay, alarm, ConditionInactive, ack, Acknowledged, greyOut, GreyedOut	Given a DisplayStyle in RegularDisplay, when the alarm of DisplayStyle is equal to ConditionInactive and the ack of DisplayStyle is not equal to Acknowledged and DisplayStyle receives a greyOut, then the DisplayStyle goes into GreyedOut

Table 11 Pre-processing ACE-6: Silencing audible alarm for a time period and possible reannunciation

Step: ChatGPT		
ACE-6’		Given an alarm is being acknowledged using the Alarm Ack button, when the button is pressed, then the audible alarm sound shall be silenced for 2 min, and the visual display of the alarm message shall remain unaffected. If the alarm condition ceases within the 2-minute silence period, then the alarm shall be dismissed. If the originating condition remains valid or reasserts itself after the silence period, then the alarm shall be reannunciated.
Step: Refinement		
REQ ID	**Contextual information**	**Description**
REQ−6.1	Acknowledgement, WaitForAcknowledgementViewed, silence, Sound, Acknowledged	Given an Acknowledgement in WaitForAcknowledgementViewed, when the Acknowledgement receives a silence, then the Acknowledgement sends a silence to the Sound and goes into Acknowledged
REQ−6.2	Sound, AlarmToneActiveOn, silence, AlarmToneInactive, setTimer(2min)	Given a Sound in AlarmToneActiveOn, when the Sound receives a silence, then the Sound calls setTimer with 2min and the Sound goes into AlarmToneInactive
REQ−6.3	Sound, AlarmToneInactive, timer, alarm, ConditionActive, Acknowledgement, stopSilence, AlarmToneActiveToneOff	Given a Sound in AlarmToneInactive, when the timer of the Sound is equal to 2 min and the alarm of the Sound is equal to ConditionActive, then the Sound sends a stopSilence to the Acknowledgement and the Sound goes into AlarmToneActiveToneOff
REQ−6.4	Acknowledgement, Acknowledged, stopSilence, WaitForAcknowledgementNotViewed	Given an Acknowledgement in Acknowledged, when the Acknowledgement receives a stopSilence, then the Acknowledgement goes into WaitForAcknowledgementNotViewed

requirement detection does not produce any further refinements. Requirement REQ−6.1 matches the meta-requirement `<G1,W1,T7 and T2>` (Table 17), and it is a refinement of the previous meta-requirement. Therefore, the translation of this meta-requirement refines the already created transition by adding the effect.

Observe that the order of translation of the requirements involving the same transition does not affect the results.

Mapping of multiple alarm occurrence requirement. If REQ−2.1 is also considered (Table 13), the resulting state machine of DisplayStyle component does not behave as intended (Fig. 17). Indeed, the occurrence of the internal transition (added by translating the instantiated requirement patterns of REQ−2.1) and of the outgoing transition (from GreyedOut to RegularDisplay) is not deterministic since the two guards are not in mutex. This not intended behaviour

Table 12 Mapping REQ−1.1-REQ−1.4: Alarming Condition Occurrence

REQ−1.1	MetaReq (<G1,W1,T7 and T2>)
Given an AlarmingCondition in ConditionInactive	Given a <<Block as gContext>> in <<State as gSource>>
when the AlarmingCondition receives a conditionActivate	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
then the AlarmingCondition sends a regular event to DisplayStyle	Then the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
and	and
the AlarmingCondition goes into ConditionActive	the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	**Role**
AlarmingCondition	Block as: gContext, wReceiver, tSender, tContext
ConditionInactive	State as gSource
conditionActivate	Signal as wEvent
regular	Signal as tMessage
DisplayStyle	Block as tReceiver
ConditionActive	State as tTarget
REQ−1.2	**MetaReq (<G1,W1,T7 and T2>)**
Given a DisplayStyle in GreyedOut	Given a <<Block as gContext>> in <<State as gSource>>
when the DisplayStyle receives a regular event	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
then the DisplayStyle sends an alarmDisplay to Acknowledgement	Then the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
and	and
the DisplayStyle goes into RegularDisplay	the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	**Role**
DisplayStyle	Block as: gContext, wReceiver, tSender, tContext
GreyedOut	State as gSource
regular	Signal as wEvent
alarmDisplay	Signal as tMessage
Acknowledgement	Block as tReceiver
RegularDisplay	State as tTarget
REQ−1.3	**MetaReq (<G1,W1,T1 and T2>)**
Given a Sound in AlarmToneInactive	Given a <<Block as gContext>> in <<State as gSource>>
when the Sound receives a conditionActivate	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
then the Sound calls setAlarmTone with ON	Then the <<Block as tContext1>> <<Operation as tAction>>
and	and

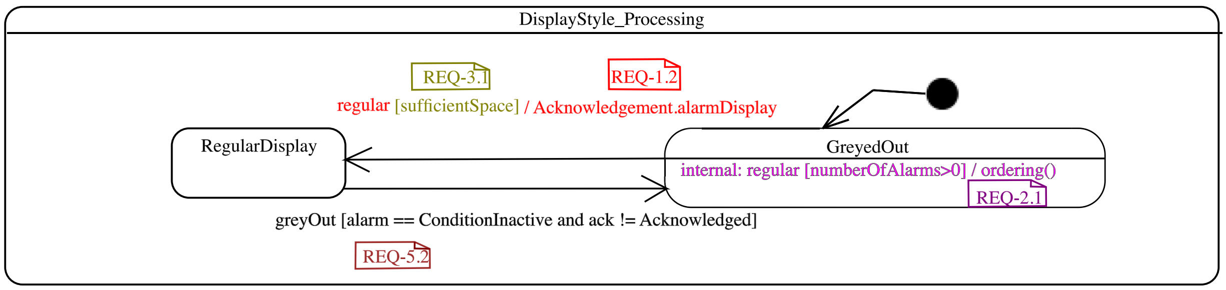

DisplayStyle_Processing
REQ-3.1
REQ-1.2
regular [sufficientSpace] / Acknowledgement.alarmDisplay
RegularDisplay
GreyedOut
internal: regular [numberOfAlarms>0] / ordering()
REQ-2.1
greyOut [alarm == ConditionInactive and ack != Acknowledged]
REQ-5.2

Fig. 17 Behavioural view: state machines of DisplayStyle considering refined requirement REQ−2.1

Table 12 continued

REQ−1.3	MetaReq (<G1,W1,T1 and T2>)
the Sound goes into AlarmToneActiveToneOn	the <<Block as tContext2>> goes into <<State as tTarget>>
Role Matching Value	Role
Sound	Block as: gContext, wReceiver, tContext1, tContext2
AlarmToneInactive	State as gSource
conditionActivate	Signal as wEvent
alarmingTone(ON)	Operation as tAction
AlarmToneActiveToneOn	State as tTarget
REQ−1.4	MetaReq (<G1,W1,T2>)
Given an Acknowledgement in wait for WaitForAcknowledgementNotViewed	Given a <<Block as gContext>> in <<State as gSource>>
when the Acknowledgement receives an alarmDisplay	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
then the Acknowledgement goes in WaitForAcknowledgementViewed	Then the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	Role
Acknowledgement	Block as: gContext, wReceiver, tContext
WaitForAcknowledgementNotViewed	State as gSource
alarmDisplay	Signal as wEvent
WaitForAcknowledgementViewed	State as tTarget

Table 13 Mapping REQ−2.1: Multiple Alarms Occurrence

REQ−2.1	MetaReq (<G1, W1 and W3, T1>)
Given a DisplayStyle in GreyedOut	Given a <<Block as gContext>> in <<State as gSource>>
when DisplayStyle receives a regular event	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
and	and
the number of alarms of DisplayStyle is greater than zero	the <<Property as wVar>> of <<Block as wContext>> <<comp_op>> <<ValueSpecification as wValue>>
then the DisplayStyle orders the alarms (by severity first, then in order of occurrence, newest first)	Then the <<Block as tContext>> <<Operation as tAction>>
Role Matching Value	Role
DisplayStyle	Block as: gContext, wReceiver, wContext, tContext
GreyedOut	State as gSource
regular	Signal as wEvent
numberOfAlarms	Property as wVar
greater than	comp_op
zero	ValueSpecification as wValue
ordering	Operation as tAction

Table 14 Mapping REQ−3.1-REQ−3.2: Not Displayed Alarm

REQ−3.1	MetaReq (<G1, W1 and W3, T7 and T2>)
Given a DisplayStyle in GreyedOut	Given a <<Block as gContext>> in <<State as gSource>>
when the DisplayStyle receives a regular event	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
and	and
the sufficient space of DisplayStyle is true	the <<Property as wVar>> of <<Block as wContext>> <<comp_op>> <<ValueSpecification as wValue>>
then the DisplayStyle sends an alarmDisplay to Acknowledgement	Then the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
and	and
the DisplayStyle goes into RegularDisplay	the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	**Role**
DisplayStyle	Block as: gContext, wReceiver, wContext, tSender, tContext
GreyedOut	State as gSource
regular	Signal as wEvent
sufficientSpace	Property as wVar
is (equal to)	<<comp_op>>
true	ValueSpecification as wValue1
alarmDisplay	Signal as tMessage
Acknowledgement	tReceiver
RegularDisplay	State as tTarget
REQ−3.2	**MetaReq (<G1, W1, T2>)**
Given an Acknowledgement in WaitForAcknowledgementViewed	Given a <<Block as gContext>> in <<State as gSource>>
When the Acknowledgement receives a silence	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
Then the Acknowledgement goes in Acknowledged	Then the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	**Role**
Acknowledgement	Block as: gContext, wReceiver, tContext
WaitForAcknowledgementViewed	State as gSource
silence	Signal as wEvent
Acknowledged	State as tTarget

could detected by model checking the state machine and manually corrected.

A.4 Evaluation

We evaluate the effectiveness of the approach by comparing the transitions of the resulting SM models and the model elements (transitions and states) annotated with *ACE-n* in the original model [18].

Table 18 summarizes the mapping of the transitions and places–annotated with requirements *ACE-n*–in the original state machine to the transitions—annotated with the derived requirements *REQ-n.m*—in the derived model of Fig. 15. In the following, we consider each concurrent region in Fig. 13–respectively, each state machine in Fig. 15 and, informally, discuss about the semantic equivalence of the mapping.

AlarmingCondition_Status There is a one-to-one mapping between the transitions in the concurrent region and those in the corresponding state machine—see Table 18, rows 1 and 2. The derived transitions are behavioural equivalent to the mapped transitions: in particular, the effects produce the same events (trigger event of DisplayStyle_Processing)—regular (row 1) and greyOut (row 2).

Sound_Processing Considering rows 3 and 4 (Table 18), the transitions in the original model have been mapped to a unique transition in the derived model, which has a similar (but not the same) semantics of the former. Indeed, while

Table 15 Mapping REQ−4.1-REQ−4.2: Explicit Acknowledgement Alarm

REQ−4.1	MetaReq (<G1, W2, T7>)
Given an Alarm Ack button in alarm occurred	Given a <<Block as gContext>> in <<State as gSource>>
When the user presses the Alarm Ack button	When the <<Block as wContext>> <<Operation as wAction>>
Then the Alarm Ack button sends a silence to Acknowledgement	Then the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
Role Matching Value	Role
silence	Signal as tMessage
Acknowledgement	Block as tReceiver
REQ−4.2	MetaReq (<G1, W1, T2>)
Given an Acknowledgement in WaitForAcknowledgementViewed	Given a <<Block as gContext>> in <<State as gSource>>
When the Acknowledgement receives a silence	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
Then the Acknowledgement goes into Acknowledged	Then the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	Role
Acknowledgement	Block as: gContext, wReceiver, tContext
WaitForAcknowledgementViewed	State as gSource
silence	Signal as wEvent
Acknowledged	State as tTarget

Table 16 Mapping REQ−5.1-REQ−5.2: Alarm Condition Ceases

REQ−5.1	MetaReq (< G1, W1, T7 and T2>)
Given an AlarmingCondition in ConditionActive	Given a <<Block as gContext>> in <<State as gSource>>
when the AlarmingCondition receives a conditionCeases	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
then the AlarmingCondition sends a greyOut to DisplayStyle	Then the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
and	and
the AlarmingCondition goes into ConditionInactive	the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	Role
AlarmingCondition	Block as: gContext, wReceiver, tSender, tContext
ConditionActive	State as gSource
conditionCeases	Signal as wEvent
DisplayStyle	Block as tReceiver
greyOut	Signal as tMessage
ConditionInactive	State as tTarget

the trigger "conditionActivate"—row 3—is present in the derived transition, the time-out trigger "tm(IN_TIME)"—row 4—is not; the reason is that no information about the time-out is given in the original requirement *ACE-1*, and therefore in the derived requirement *REQ−1.3*.

The original transitions, in rows 5 and 6, have been mapped to more informative transitions in the derived model: indeed, the latter are characterized by the same effect as the former and, additionally, by an effect (i.e. the time-out setting to 2 min) which was specified in the refined requirement *REQ−6.2* but not in the original one *ACE-6*.

The original transition in row 7 has been mapped to a transition with similar semantics in the derived model; the time-out trigger "tm(SILENCE_TIME)" is specified as part

Table 16 continued

REQ-5.2	MetaReq (<G1, W3 and W3 and W1, T2>)
Given a DisplayStyle in RegularDisplay,	Given a <<Block as gContext>> in <<State as gSource>>,
when the alarm of DisplayStyle is equal to ConditionInactive	When the <<Property as wVar1>> of <<Block as wContext1>> <<comp_op1>> <<ValueSpecification as wValue1>>
and	and
the ack of DisplayStyle is not equal to Acknowledged	the <<Property as wVar2>> of <<Block as wContext2>> <<comp_op2>> <<ValueSpecification as wValue2>>
and	and
DisplayStyle receives a greyOut,	the <<Block as wReceiver>> receives <<Signal as wEvent>>,
Then the DisplayStyle goes into GreyedOut.	Then the <<Block as tContext>> goes into <<State as tTarget>>

Role Matching Value	Role
DisplayStyle	Block as: gContext, wContext1, wContext2, wReceiver, tContext
RegularDisplay	State as gSource
alarm	Property as wVar1
is equal to	comp_op1
ConditionInactive	ValueSpecification as wValue1
ack	Property as wVar2
is not equal to	comp_op2
Acknowledged	ValueSpecification as wValue2
greyOut	Signal as wEvent
GreyedOut	State as tTarget

of the guard (“timer == 2min”) in the mapped transition. Observe that the guard also includes a further restriction (“alarm == ConditionActive”) which is not explicitly specified in the original model but it is stated in the original requirement *ACE-6* and in the derived requirement *REQ-6-3*. The effect of the original transition is semantically equivalent to the effect in the derived transition.

DisplayStyle_Processing A one-to-one mapping exists between the transitions in the concurrent region and those in the corresponding state machine. However, the derived transitions are more informative than the mapped transitions: indeed they have the same trigger as the corresponding transitions in the original model—“regular” (row 8) and “greyOut” (row 10)—but, additionally, they are characterized by effects and/or guards. In particular, the requirement *REQ−3.1* enables to add the guard to the transition from GreyedOut to RegularDisplay (row 10).

Acknowledgement_Processing The transitions, in the derived model, indicated in rows 11 and 14, are semantically equivalent to the corresponding mapped transitions in the original model. Rows 12 and 13 both concern the transition from WaitForAckViewed to Acknowledged, in the derived model, which is more informative than the state (NotViewed, row 12) and the transition (from Viewed to Acknowledged, row 13) in the original model. In particular, the derived transition is characterized by the same trigger as in the mapped transition (row 13) but, additionally, an effect is added—by translating the refined requirement *REQ−6.1*—which explicitly represents the cause–effect relation between the Acknowledgement and the Sound components.

Table 17 Mapping REQ−6.1-REQ−6.4: Silencing audible alarm for a time period and possible reannunciation

REQ−6.1	MetaReq (<G1, W1, T7 and T2>)
Given an Acknowledgement in WaitForAcknowledgementViewed	Given a <<Block as gContext>> in <<State as gSource>>
when the Acknowledgement receives a silence	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
then the Acknowledgement sends a silence to the Sound	Then the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
and	and
the Acknowledgement goes into Acknowledged	the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	Role
Acknowledgement	Block as: gContext, wReceiver, tSender, tContext
WaitForAcknowledgementViewed	State as gSource
silence	Signal as: wEvent, tMessage
Sound	Block as tReceiver
Acknowledged	State as tTarget
REQ−6.2	MetaReq (<G1, W1, T1 and T2>)
Given a Sound in AlarmToneActiveToneOn	a <<Block as gContext>> in <<State as gSource>>
when the Sound receives a silence	When the <<Block as wReceiver>> receives <<Signal as wEvent>>
then the Sound calls setTimer with 2min	the <<Block as tContext1>> <<Operation as tAction>>
and	and
the Sound goes into AlarmToneInactive	Then, the <<Block as tContext2>> goes into <<State as tTarget>>
Sound	Block as: gContext, wReceiver, tContext1, tContext2
AlarmToneActiveToneOn	State as gSource
silence	Signal as wEvent
AlarmToneInactive	State as tTarget
setTimer(2min)	Operation as tAction
REQ-6.3	MetaReq (<G1, W3 and W3, T7 and T2>)
Given a Sound in AlarmToneInactive,	Given a <<Block as gContext>> in <<State as gSource>>,
when the timer of the Sound is equal to 2 minutes	When the <<Property as wVar1>> of <<Block as wContext1>> <<comp_op1>> <<ValueSpecification as wValue1>>
and	and
the alarm of the Sound is equal to ConditionActive,	the <<Property as wVar2>> of <<Block as wContext2>> <<comp_op2>> <<ValueSpecification as wValue2>>,
then the Sound sends a stopSilence to the Acknowledgement	Then, the <<Block as tSender>> sends a <<Signal as tMessage>> to <<Block as tReceiver>>
and	and
the Sound goes into AlarmToneActiveToneOff.	the <<Block as tContext>> goes into <<State as tTarget>>

Table 17 continued

Role Matching Value	Role
Sound	Block as: gContext, wContext1, wContext2, tSender, tContext
AlarmToneInactive	State as gSource
timer	Property as wVar1
is equal to	comp_op1
2 minutes	ValueSpecification as wValue1
alarm	Property as wVar2
is equal to	comp_op2
ConditionActive	ValueSpecification as wValue2
stopSilence	Signal as tMessage
Acknowledgement	Block as tReceiver
AlarmToneActiveToneOff	State as tTarget
REQ-6.4	MetaReq (<G1, W1, T2>)
Given an Acknowledgement in Acknowledged,	Given a <<Block as gContext>> in <<State as gSource>>
when the Acknowledgement receives a stopSilence,	When the <<Block as wReceiver>> receives <<Signal as wEvent>>,
then the Acknowledgement goes into WaitForAcknowledgementNotViewed.	Then, the <<Block as tContext>> goes into <<State as tTarget>>
Role Matching Value	Role
Acknowledgement	Block as: gContext, wReceiver, tContext
Acknowledged	State as gSource
stopSilence	Signal as wEvent
WaitForAcknowledgementNotViewed	State as tTarget

Table 18 Mapping transitions and places from the original model (O) to the derived model (D)

AlarmingCondition_Status							
N	M	Req	Source	Target	Trigger	Effect	Guard
1	O	ACE-1	ConditionInactive	ConditionActive	conditionActivate	GEN(regular)	
	D	REQ−1.1	ConditionInactive	ConditionActive	conditionActivate	DS.regular	
2	O	ACE-5	ConditionActive	ConditionInactive	conditionCeases	GEN(greyOut)	
	D	REQ−5.1	ConditionActive	ConditionInactive	conditionCeases	DS.greyOut	
Sound_Processing							
	M	Req	Source	Target	Trigger	Effect	Guard
3	O	ACE-1	AlarmToneInactive	AlarmToneActive	conditionActivate		
	D	REQ−1.3	AlarmToneInactive	AlarmToneActiveToneOn	conditionActivate	setAlarmTone(ON)	
4	O	ACE-1	ToneOff	ToneOn	tm(IN_TIME)	sound(ON)	
	D	REQ−1.3	AlarmToneInactive	AlarmToneActiveToneOn	conditionActivate	setAlarmTone(ON)	
5	O	ACE-6	AlarmToneActive	AlarmToneInactive	silence		
	D	REQ−6.2	AlarmToneActiveToneOn	AlarmToneInactive	silence	setTimer(2min)	
	D	REQ−6.2	AlarmToneActiveToneOff	AlarmToneInactive	silence	setTimer(2min)	
6	O	ACE-6	AlarmToneInactive	AlarmToneInactive	silence		
	D	REQ−6.2	AlarmToneInactive	AlarmToneInactive	silence	setTimer(2min)	
7	O	ACE-6	AlarmToneInactive	AlarmToneActive	tm(SILENCE_TIME)	GEN(stopSilence)	
	D	REQ−6.3	AlarmToneInactive	AlarmToneActiveToneOff		Ack.stopSilence	timer == 2min and alarm == ConditionActive
DisplayStyle_Processing							
	M	Req	Source	Target	Trigger	Effect	Guard
8	O	ACE-1	GreyedOut	RegularDisplay	regular		

Table 18 continued

DisplayStyle_Processing							
	M	Req	Source	Target	Trigger	Effect	Guard
	D	REQ1.2	GreyedOut	RegularDisplay	regular	Ack.alarmDisplay	
9	O	ACE-3	state: RegularDisplay				
	D	REQ−3.1	GreyedOut	RegularDisplay	regular	Ack.alarmDisplay	sufficientSpace
10	O	ACE-5	RegularDisplay	GreyedOut	greyOut		
	D	REQ−5.2	RegularDisplay	GreyedOut	greyOut		alarm == ConditionInactive and ack != Acknowledged

Acknowledgement_Processing							
	M	Req	Source	Target	Trigger	Effect	Guard
11	O	ACE-1	NotViewed	Viewed	alarmDisplay		
	D	REQ−1.4	WaitForAckNotViewed	WaitForAckViewed	alarmDisplay		
12	O	ACE-3	state: NotViewed				
	D	REQ−3.2	WaitForAckViewed	Acknowledged	silence		
13	O	ACE-4	Viewed	Acknowledged	silence		
	D	REQ−4.1, REQ−6.1	WaitForAckViewed	Acknowledged	silence	Sound.silence	
14	O	ACE-6	Acknowledged	NotViewed	stopSilence		
	D	REQ−6.4	Acknowledged	WaitForAckNotViewed	stopSilence		

References

1. Aceituna, D., Do, H., Walia, G.S., Lee, S.W.: Evaluating the use of model-based requirements verification method: a feasibility study (2011). https://doi.org/10.1109/EmpiRE.2011.6046248
2. Alferez, M., Pastore, F., Sabetzadeh, M., Briand, L., Riccardi, J.R.: Bridging the gap between requirements modeling and behavior-driven development. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 239–249. IEEE (2019)
3. Bankauskaite, J., Morkevicius, A.: An approach: SysML-based automated completeness evaluation of the system requirements specification. pp. 66–72 (2018)
4. Basin, D., Clavel, M., Egea, M.: A decade of model-driven security. In: Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT, pp. 1–10 (2011). https://doi.org/10.1145/1998441.1998443
5. Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional, Boston (2003)
6. Bernardi, S., Merseguer, J., Petriu, D.C.: Dependability modeling and analysis of software systems specified with uml. ACM Comput. Surv. **45**(1) (2012). https://doi.org/10.1145/2379776.2379778
7. Bernardi, S., Merseguer, J., Petriu, D.C.: Model-Driven Dependability Assessment of Software Systems. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-39512-3
8. de Biase, M.S., Marrone, S., Palladino, A.: Towards Automatic Model Completion: from Requirements to SysML State Machines. arXiv preprint arXiv:2210.03388 (2022). https://doi.org/10.48550/arXiv.2210.03388
9. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, Second Edition. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers (2017). https://doi.org/10.2200/S00751ED2V01Y201701SWE004
10. Bruel, J.M., Ebersold, S., Galinier, F., Mazzara, M., Naumchev, A., Meyer, B.: The role of formalism in system requirements. ACM Comput. Surv **54** (2021). https://doi.org/10.1145/3448975
11. Brunello, A., Montanari, A., Reynolds, M.: Synthesis of LTL formulas from natural language texts: State of the art and research directions. In: J. Gamper, S. Pinchinat, G. Sciavicco (eds.) Leibniz International Proceedings in Informatics, LIPIcs, pp. 171–1719. Germany (2019). https://doi.org/10.4230/LIPIcs.TIME.2019.17
12. Burgueño, L., Clarisó, R., Gérard, S., Li, S., Cabot, J.: An NLP-based architecture for the autocompletion of partial domain models. In: International Conference on Advanced Information Systems Engineering (CAiSE 2021), pp. 91–106. Springer (2021)
13. Buzhinsky, I.: Formalization of natural language requirements into temporal logics: a survey. In: 2019 IEEE 17th international conference on industrial informatics (INDIN), vol. 1, pp. 400–406. IEEE (2019)
14. Colombo, P., Khendek, F., Lavazza, L.: Bridging the gap between requirements and design: An approach based on problem frames and sysml. J. Syst. Softw. **85**(3), 717–745 (2012). https://doi.org/10.1016/j.jss.2011.09.046
15. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis. Springer, Berlin Heidelberg (2011)
16. Dalpiaz, F., Ferrari, A., Franch, X., Palomares, C.: Natural language processing for requirements engineering: The best is yet to come. IEEE Softw. **35**(5), 115–119 (2018). https://doi.org/10.1109/MS.2018.3571242
17. Dimanidis, A., Chatzidimitriou, K.C., Symeonidis, A.L.: A Natural Language Driven Approach for Automated Web API Development: Gherkin2OAS. WWW '18. In: Companion Proceedings of The Web Conference 2018. International World Wide Web Conferences Steering Committee (2018). https://doi.org/10.1145/3184558.3191654
18. Douglass, B.P.: Real-time UML - developing efficient objects for embedded systems. Addison-Wesley object technology series, Addison-Wesley-Longman (1998)
19. Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., Moreschini, P.: Assisting requirement formalization by means of natural language translation. Formal Methods in System Design **4**, 243–263 (1994). https://doi.org/10.1007/BF01384048
20. Ferrari, A., Dell'Orletta, F., Spagnolo, G.O., Gnesi, S.: Measuring and improving the completeness of natural language requirements. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **8396 LNCS**, 23 - 38 (2014). https://doi.org/10.1007/978-3-319-05843-6_3
21. Flammini, F., Marrone, S., Iacono, M., Mazzocca, N., Vittorini, V.: A Multiformalism Modular Approach to ERTMS/ETCS Failure Modelling, vol. 21. International Journal of Reliability, Quality and Safety Engineering (2014)https://doi.org/10.1142/S0218539314500016
22. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE '07), pp. 37–54 (2007). https://doi.org/10.1109/FOSE.2007.14
23. Frederiksen, S.J., Aromando, J., Hsiao, M.S.: Automated assertion generation from natural language specifications. In: 2020 IEEE International Test Conference (ITC), pp. 1–5. IEEE (2020)
24. Furness, N., van Houten, H., Arenas, L., Maarten, B.: ERTMS level 3: the game-changer (2017)
25. Hesenius, M., Griebe, T., Gruhn, V.: Towards a behavior-oriented specification and testing language for multimodal applications. In: Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, p. 117-122. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2607023.2610278
26. Horváth, B., Molnár, V., Graics, B., Hajdu, Á., Ráth, I., Horváth, Á., Karban, R., Trancho, G., Micskei, Z.: Pragmatic verification and validation of industrial executable sysml models. Systems Engineering (2023)
27. Hähnle, R., Johannisson, K., Ranta, A.: An authoring tool for informal and formal requirements specifications (2002). https://doi.org/10.1007/3-540-45923-5_16
28. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Sci. Comput. Program. **72**, 31–39 (2008). https://doi.org/10.1016/j.scico.2007.08.002
29. Kamalakar, S., Edwards, S.H., Dao, T.M.: Automatically generating tests from natural language descriptions of software behavior. In: Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,, pp. 238–245. INSTICC, SciTePress (2013). https://doi.org/10.5220/0004566002380245
30. Kassab, M., Neill, C., Laplante, P.: State of practice in requirements engineering: Contemporary data. Innov. Syst. Softw. Eng. **10**(4), 235–241 (2014). https://doi.org/10.1007/s11334-014-0232-4
31. Kolahdouz-Rahimi, S., Lano, K., Lin, C.: Requirement formalisation using natural language processing and machine learning: A systematic review. In: Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering, MODELSWARD 2023, Lisbon, Portugal, February 19-21, pp. 237–244. SCITEPRESS (2023). https://doi.org/10.5220/0011789700003402
32. Konrad, S., Cheng, B.H.: Automated analysis of Natural Language properties for UML models. International Conference on Model Driven Engineering Languages and Systems (2005)
33. Kuhn, T.: A survey and classification of controlled natural languages. Comput. Linguist. **40**(1), 121–170 (2014). https://doi.org/10.1162/COLI_a_00168
34. Lenka, R.K., Kumar, S., Mamgain, S.: Behavior driven development: Tools and challenges. In: 2018 International Confer-

ence on Advances in Computing, Communication Control and Networking (ICACCCN), pp. 1032–1037 (201https://doi.org/10.1109/ICACCCN.2018.8748595
35. Li, F.L., Horkoff, J., Borgida, A., Guizzardi, G., Liu, L., Mylopoulos, J.: From stakeholder requirements to formal specifications through refinement (2015). https://doi.org/10.1007/978-3-319-16101-3_11
36. Li, N., Escalona, A., Kamal, T.: Skyfire: Model-based testing with Cucumber. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 393–400 (2016)https://doi.org/10.1109/ICST.2016.41
37. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77** (1989). https://doi.org/10.1109/5.24143
38. Mussbacher, G., Combemale, B., Kienzle, J., Abrahão, S., Ali, H., Bencomo, N., Búr, M., Burgueño, L., Engels, G., Jeanjean, P., Jézéquel, J.M., Kühn, T., Mosser, S., Sahraoui, H., Syriani, E., Varro, D., Weyssow, M.: Opportunities in intelligent modeling assistance. Softw. Syst. Model. (2020). https://doi.org/10.1007/s10270-020-00814-5
39. (OMG), O.M.G.: OMG Unified Modeling Language (UML) Superstructure Version 2.4.1 (2011). http://www.omg.org/spec/UML/2.4.1/Superstructure
40. (OMG), O.M.G.: OMG Systems Modeling Language (OMG SysML^{TM}) Version 1.6 (2019). https://www.omg.org/spec/SysML/1.6/
41. OutSystems: Outsystems (2022). https://www.outsystems.com/p/low-code-platform/
42. Pereira, L., Sharp, H., de Souza, C., Oliveira, G., Marczak, S., Bastos, R.: Behavior-driven development benefits and challenges: reports from an industrial study. XP ’18. ACM (2018). https://doi.org/10.1145/3234152.3234167
43. Ramackers, G.J., Griffioen, P.P., Schouten, M.B., Chaudron, M.R.: From prose to prototype: Synthesising executable UML models from natural language. 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) (2021)
44. Richardson, L., Ruby, S.: RESTful Web Services. O’Reilly Media Inc, Sebastopol (2008)
45. Rocha Silva, T.: Towards a domain-specific language to specify interaction scenarios for web-based graphical user interfaces. In: Companion of the 2022 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS ’22 Companion, pp. 48–53. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3531706.3536463
46. Saini, R.: Automated, traceable, and interactive domain modelling. pp. 217–220 (2022). https://doi.org/10.1145/3550356.3552372
47. Saini, R., Mussbacher, G., Guo, J.L., Kienzle, J.: Automated, interactive, and traceable domain modelling empowered by artificial intelligence. Softw. Syst. Model. **21**(3), 1015–1045 (2022)
48. Sayar, I., Souquières, J.: Bridging the gap between requirements document and formal specifications using development patterns. In: 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW), pp. 116–122 (2019). https://doi.org/10.1109/REW.2019.00026
49. Smart, J.F., Molak, J.: BDD in Action: Behavior-driven development for the whole software lifecycle. Simon and Schuster (2023)
50. Snook, C., Hoang, T.S., Dghyam, D., Butler, M., Fischer, T., Schlick, R., Wang, K.: Behaviour-driven formal model development. In: Sun, J., Sun, M. (eds.) Formal Methods and Software Engineering, pp. 21–36. Springer International Publishing, Cham (2018)
51. Whittle, J., Sawyer, P., Bencomo, N., Chengy, B.H., Bruelz, J.M.: Relax: Incorporating uncertainty into the specification of self-adaptive systems (2009). https://doi.org/10.1109/RE.2009.36
52. Wiecher, C., Japs, S., Kaiser, L., Greenyer, J., Dumitrescu, R., Wolff, C.: Scenarios in the loop: Integrated requirements analysis and automotive system validation. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS ’20. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3417990.3421264
53. Wynne, M., Hellesøy, A., Tooke, S.: The Cucumber Book: Behaviour-driven Development for Testers and Developers. Pragmatic programmers. Pragmatic Bookshelf (2017). https://books.google.it/books?id=zzOqnQAACAAJ
54. Yang, S., Sahraoui, H.: Towards automatically extracting UML class diagrams from natural language specifications. pp. 396–403 (2022). https://doi.org/10.1145/3550356.3561592
55. Yang, S., Zhao, Y., Tu, K.: PCFGS can do better: inducing probabilistic context-free grammars with many symbols. pp 1487–1498 (2021)
56. Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K.J., Ajagbe, M.A., Chioasca, E.V., Batista-Navarro, R.T.: Natural language processing for requirements engineering: a systematic mapping study. ACM Comput. Surv. **54**(3), 1–41 (2021). https://doi.org/10.1145/3444689

Maria Stella de Biase is a researcher in the Department of Mathematics at Universitá degli Studi della Campania “L. Vanvitelli”. She holds a MS degree in mathematics and a Ph.D. degree in Mathematics, Physics, and Engineering Applications both from the Universitá degli Studi della Campania “L. Vanvitelli”. Her expertise spans software engineering, natural language processing, and climate change resilience, with a focus on multidisciplinary approaches. The research she is conducting focuses about advancing and improving the integration of artificial intelligence in system design. She has contributed to several international conferences and holds certifications in Python for data science, NLP, and UML fundamentals.

Simona Bernardi is an Assistant Professor in the Department of Computer Science and Systems Engineering at the University of Zaragoza, Spain. She received a MS degree in mathematics and a PhD degree in computer science, in 1997 and 2003, respectively, both from the University of Torino, Italy. Her research interests are in the area of software engineering, in particular model driven engineering, verification and validation of performance, dependability and survivability software requirements, and formal methods for the modelling and analysis of software systems. She is also interested in the application of anomaly detection techniques in energy frauds. She has served as a referee for international journals and as a program committee member for several international conferences and workshops.

Stefano Marrone is an associate professor in Computer Science at Universit' della Campania "Luigi Vanvitelli", Italy. His interests include the definition of model-driven processes for the design and the analysis of transportation control systems, complex communication networks and critical infrastructures. He is involved in research projects with both academic and industrial partners. He co-authored more than 120 papers in international journals and conference proceedings.

José Merseguer is a Full Professor of Software Engineering with the Department of Computer Science and Systems Engineering at the University of Zaragoza, Spain. He was the director of the PhD Program in Computer Science and of the Computer Science Master, both at the University of Zaragoza. His main research interests include performance and dependability analysis of software systems, he served as Program Chair for ICPE. He developed postdoctoral research with Prof. M. Woodside and Prof. D. Petriu at Carleton University, Canada, with Prof. R. Lutz at Iowa State University, USA. He has been a Visiting Researcher with the University of Turin, with the University of Cagliari and with the Politecnico di Milano, Italy. He is a co-author of the book 'Model-driven dependability assessment of software-systems,' Springer, and has advised three PhD thesis.

Angelo Palladino is head of Aerospace business unit in Kineton srl-Societ' Benefit. He has developed consulting activities for different companies as projects technical responsible in aerospace and automotive fields. The main expertise have been developed as business manager and project manager for the development and application of innovative methodologies in the propulsion modelling and electronic control systems design. The consulting topics go from the spark ignition internal combustion engine modelling and control to the three-way catalytic converters modelling and identification, to the hybrid vehicles. During the consulting activities are been used advanced techniques about automotive fields, including control strategies for delayed systems, "receding horizon" algorithms and "sliding mode" technique, order reduction techniques, genetic algorithms for parameters identification, neural network, Hardware-In-the-Loop testing environment system and co-simulation application, Petri Net technicques organization and optimization. Several of these topics have been inspired by collaborations with research centers of companies such as AnsaldoBreda, Magneti Marelli, Centro Ricerche Fiat, General Motors and Fiat Powertrain Technologies. On May 2006, he got a PhD title in collaboration with Fiat Powertrain Technologies - ELASIS FIAT Research Group on the NARMAX identification and engine calibration.