# Automatic Generation of Acceptance Test Cases from Use Case Specifications: an NLP-based Approach

Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand

**Abstract**—Acceptance testing is a validation activity performed to ensure the conformance of software systems with respect to their functional requirements. In safety critical systems, it plays a crucial role since it is enforced by software standards, which mandate that each requirement be validated by such testing in a clearly traceable manner. Test engineers need to identify all the representative test execution scenarios from requirements, determine the runtime conditions that trigger these scenarios, and finally provide the input data that satisfy these conditions. Given that requirements specifications are typically large and often provided in natural language (e.g., use case specifications), the generation of acceptance test cases tends to be expensive and error-prone.

In this paper, we present Use Case Modeling for System-level, Acceptance Tests Generation (UMTG), an approach that supports the generation of executable, system-level, acceptance test cases from requirements specifications in natural language, with the goal of reducing the manual effort required to generate test cases and ensuring requirements coverage. More specifically, UMTG automates the generation of acceptance test cases based on use case specifications and a domain model for the system under test, which are commonly produced in many development environments. Unlike existing approaches, it does not impose strong restrictions on the expressiveness of use case specifications. We rely on recent advances in natural language processing to automatically identify test scenarios and to generate formal constraints that capture conditions triggering the execution of the scenarios, thus enabling the generation of test data. In two industrial case studies, UMTG automatically and correctly translated 95% of the use case specification steps into formal constraints required for test data generation; furthermore, it generated test cases that exercise not only all the test scenarios manually implemented by experts, but also some critical scenarios not previously considered.

**Index Terms**—System Test Case Generation; Use Case Specifications; Natural Language Processing; Semantic Role Labeling

## 1 Introduction

The complexity of embedded software in safety critical domains, e.g., automotive and avionics, has significantly increased over the years. In such contexts, software testing plays a fundamental role to ensure that the system, as a whole, meets its functional requirements. Such system testing activity is commonly referred to as *acceptance testing* [1]. As opposed to verification, which aims at detecting faults, acceptance testing is a validation activity performed at the very end of the development lifecycle to demonstrate compliance with requirements. Acceptance testing is enforced and regulated by international standards. For example, traceability between requirements and acceptance test cases is enforced by standards for safety-critical embedded systems [2], [3].

Functional requirements in safety critical domains are often expressed in natural language (NL), and acceptance test cases are either manually derived from requirements specifications or – a much less frequent practice – generated from models produced for testing purposes only [4], [5], [6], [7]. Automatic test generation from requirements specifications in NL achieves the best of both worlds as it does not require additional test modeling, reduces the cost of testing and also guarantees that testing is systematic and properly covers all requirements.

The benefits of automatic test generation are widely acknowledged today and there are many proposed approaches in the literature [8]. However, existing approaches are often not applicable in industrial context [9] since they typically require that system specifications be captured as UML behavioral models such as activity diagrams [10], statecharts [11], and sequence diagrams [5]. In modern industrial systems, these behavioral models tend to be complex and expensive if they are to be precise and complete enough to support test automation, and are thus often not part of development practice. There are techniques [12] [13] [14] that generate test models from NL requirements, but the generated models need to be manually edited to enable test automation, thus creating scalability issues. In approaches generating test cases directly from NL requirements [15] [16] [17] [18], test cases are not executable and often require significant manual intervention to provide test input data (e.g., they need additional formal specifications [18]). A few approaches can generate executable test cases including test input data directly from NL requirements specifications [19] [20], but they require that requirements specifications be written according to a controlled natural language (CNL). The CNL specifications are translated into formal specifications which are then used to automatically generate test input data (e.g., using

• *C. Wang, F. Pastore, A. Goknil and L.C. Briand are with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg. L.C. Briand is also affiliated with the school of EECS, University of Ottawa.*
*E-mail: wangchunhui@me.com fabrizio.pastore@uni.lu ar.goknil@gmail.com lionel.briand@uni.lu lbriand@uottawa.ca*

*Manuscript submitted to TSE*

constraint solving). The CNL language supported by these approaches is typically very focused and therefore limited (e.g., it enables the use of only a few verbs in requirements specifications), thus reducing their usability.

Our goal in this paper is to enable automated generation of executable, system-level, acceptance test cases from NL requirements, with no additional behavioral modeling. Our motivation is to rely, to the largest extent possible, on practices that are already in place in many companies developing embedded systems, including our industry partner, i.e., IEE S.A. (in the following "IEE") [21], with whom we performed the case studies reported in this paper. In many environments like IEE, development processes are use case-driven and this strongly influences their requirements engineering and acceptance testing practices. Use case specifications are widely used for communicating requirements among stakeholders and, in particular, facilitating communication with customers. A domain model typically complements use cases by specifying the terminology and concepts shared among all stakeholders and thus helps avoid misunderstandings.

In this paper, we propose, apply and assess *Use Case Modeling for System-level, Acceptance Tests Generation (UMTG)*, an approach that generates executable system test cases for acceptance testing by exploiting behavioral information in use case specifications. UMTG requires a domain model (i.e., a class diagram) of the system, which enables the generation of constraints that are used to generate test input data. Use case specifications and domain models are common in requirements engineering practice [22], such as our industry partner's organisation in our case studies. Consistent with the objectives stated above, we avoid behavioral modeling (e.g., activity and sequence diagrams) by applying Natural Language Processing (NLP) to a more structured and analysable form of use case specifications, i.e., Restricted Use Case Modeling (RUCM) [12]. Without limiting expressiveness, RUCM introduces a template with keywords and restriction rules to reduce ambiguity in requirements and enables automated analysis of use case specifications. It enables the extraction of behavioral information by reducing imprecision and incompleteness in use case specifications. RUCM has been successfully applied in many domains (e.g., [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35]). It has been previously evaluated through controlled experiments and showed to be usable and beneficial with respect to making use case specifications less ambiguous and more amenable to precise analysis and design [12]. By relying on RUCM, UMTG, differently from approaches relying on CNL, enables engineers to write specifications using the entire English vocabulary. UMTG puts limits only on the complexity of the use case specification sentences, not on the terminology in use. Keywords are only used to define a use case specification template, which is a common practice, and to introduce conditional sentences which are written in free form. In short, UMTG attempts to strike a balance among several objectives: use cases legible by all stakeholders, sufficient information for automated acceptance test cases generation, and minimal modeling.

UMTG employs NLP to build *Use Case Test Models* (UCTMs) from RUCM specifications. A UCTM captures the control flow implicitly described in an RUCM specification and enables the model-based identification of use case scenarios (i.e, the sequences of use case steps in the model). UMTG includes three model-based, coverage strategies for the generation of use case scenarios from UCTMs: branch, def-use, and subtype coverages. A list of textual pre, post and guard conditions in each use case specification is extracted during NLP. The extracted conditions enable UMTG to determine the constraints that test inputs need to satisfy to cover a test scenario. To automatically generate test input data for testing, UMTG automatically translates each extracted condition in NL into a constraint in the Object Constraint Language (OCL) [36] that describes the condition in terms of the entities in the domain model. UMTG relies on OCL since it is the natural choice for constraints in UML class diagrams. To generate OCL constraints, it exploits the capabilities of advanced NLP techniques (e.g., Semantic Role Labeling [37]). The generated OCL constraints are then used to automatically generate test input data via constraint solving using Alloy [38]. Test oracles are generated by processing the postconditions.

Engineers are expected to manually inspect the automatically generated OCL constraints, possibly make corrections and write new constraints when needed. Note that the required manual effort is very limited since, according to our industrial case studies, UMTG can automatically and correctly generate 95% of the OCL constraints. The accuracy of the OCL constraint generation is very high, since 99% of the generated constraints are correct. Executable test cases are then generated by identifying – using a mapping table – the test driver API functions to be used to provide the generated test input data to the system under test.

This paper extends our previous conference papers concerning the automatic generation of UCTMs [39] and the automatic generation of OCL constraints from specifications in NL [40] published at the International Symposium on Software Testing and Analysis (ISSTA'15) and at the 11th IEEE Conference on Software Testing, Validation and Verification (ICST'18). An earlier version of our tool was demonstrated [41] at the 10th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15). This paper brings together, refines, and extends the ideas from the above papers. Most importantly, we extend the expressiveness of the automatically generated OCL constraints by supporting existential quantifiers and size operators in addition to universal quantifiers, we introduce an Alloy-based constraint solving algorithm that solves the path conditions in OCL, and we integrate the def-use and subtype coverage strategies not presented in our previous work. Finally, the paper further provides substantial new empirical evidence to support the scalability of our approach, and demonstrates its effectiveness using two industrial case studies (i.e., automotive embedded systems sold in the US and EU markets). Our contributions include:

- UMTG, an approach for the automatic generation of executable acceptance test cases from use case specifications and a domain model, without resorting to behavioral modeling;
- an NLP technique generating test models (UCTMs) from use case specifications expressed with RUCM;

- an NLP technique generating OCL constraints from use case specifications for test input data generation;
- an algorithm combining UCTMs and constraint solving to automatically generate test input data, based on three different coverage criteria;
- a publicly available tool integrated as a plug-in for IBM DOORS and Eclipse, which generates executable acceptance test cases from use case specifications;
- two industrial case studies from which we provide credible empirical evidence demonstrating the applicability, scalability and benefits of our approach.

In this paper we focus on embedded systems. Although UMTG is a generic approach that relies on artefacts that may be available for any type of software system, it comprises solutions that have been designed specifically for embedded systems. These includes RUCM, which has only been partially evaluated with other types of systems (e.g., Web systems [31]), the NLP technique for generating OCL constraints which relies on patterns defined based on the characteristics of embedded systems, and the coverage criteria, which reflect the stringent testing thoroughness typical of safety-critical embedded systems.

This paper is structured as follows. Section 2 provides the background on the NLP techniques on which this paper builds the proposed test case generation approach. Section 3 introduces the industrial context of our case study to illustrate the practical motivations for our approach. Section 4 discusses the related work in light of our industrial needs. In Section 5, we provide an overview of the approach. From Section 6 to Section 12, we provide the details of the core technical parts of our approach. Section 13 presents our tool support for test case generation. Section 14 reports on the results of the empirical validation conducted with two industrial case studies. We conclude the paper in Section 15.

## 2 Background and Glossary

In this section, we present the background regarding the Natural Language Processing (NLP) techniques which we employ in UMTG, as well as a glossary defining the terminology used in the paper. The background on model-based testing is not presented because it is widely known to readers interested in software testing, who may further refer to books and surveys on the topic [42], [43], [44].

NLP refers to a set of procedures that extract structured information from documents written in NL. They are implemented as a pipeline that executes multiple analyses, e.g., tokenization, morphology analysis, and syntax analysis [45].

UMTG relies on five different NLP analyses: tokenization, named entity recognition, part-of-speech tagging, semantic role labeling (SRL), and semantic similarity detection. Tokenization splits a sentence into tokens based on a predefined set of rules (e.g., the identification of whitespaces and punctuation). Named entity recognition identifies and classifies named entities in a text into predefined categories (e.g., the names of cities). Part-of-speech (POS) tagging assigns parts of speech to each word in a text (e.g., noun, verb, pronoun, and adjective). SRL automatically determines the roles played by the phrases[1] in a sentence [45], e.g., the actor performing an activity. Semantic similarity detection determines the similarity between two given phrases.

Tokenization, named entity recognition, and POS tagging are well known in the software engineering community since they have been adopted by several approaches integrating NLP [46], [47], [48], [49], [50], [51]. However, none of the existing software testing approaches relies on SRL or combines SRL with semantic similarity detection.

Section 2.1 provides a brief description of SRL, while we present the basics of semantic similarity detection in Section 2.2. In Section 2.3, we present a glossary for some of the terms and concepts frequently used in the paper.

### 2.1 Semantic Role Labeling

SRL techniques are capable of automatically determining the roles played by words in a sentence. For the sentences *The system starts* and *The system starts the database*, SRL can determine that the actors affected by the actions are *the system* and *the database*, respectively. The component that is started coincides with the subject in the first sentence and with the object in the second sentence although the verb *to start* is used with active voice in both. This information cannot be captured by other NLP techniques like POS tagging or dependency parsing.

There are few SRL tools [52], [53], [54]. They are different in terms of models they adopt to capture roles. Semafor [52], [55] and Shalmaneser [53] are based on the FrameNet model, while the CogComp NLP pipeline (hereafter CNP [54]) uses the PropBank [56] and NomBank models [57], [58]. To the best of our knowledge, CNP is the only tool under active development, and is thus used in UMTG.

The tools using PropBank tag the words in a sentence with keywords (e.g., *A0*, *A1*, *A2*, *AN*) to indicate their roles. *A0* indicates who performs an action, while *A1* indicates the actor most directly affected by the action. For instance, the term *The system* is tagged with *A1* in the sentence *The system starts*, while the term *the database* is tagged with *A1* in the sentence *The system starts the database*. The other roles are verb-specific despite some commonalities, e.g., *A2* which is often used for the end state of an action.

PropBank includes additional roles which are not verb-specific (see Table 1). They are labeled with general keywords and match adjunct information in different sentences, e.g., *AM-NEG* indicating negative verbs. NomBank, instead, captures the roles of nouns, adverbs, and adjectives in noun phrases. It uses the same keywords adopted by PropBank. For instance, using ProbBank, we identify that the noun phrase *the watchdog counter* plays the role *A1* in the sentence *The system resets the watchdog counter*. Using NomBank, we obtain complementary information indicating the term *counter* is the main noun (tagged with *A0*), and the term *watchdog* is an attributive noun (tagged with *A1*).

PropBank does not help identify two different sentences describing similar concepts. In the sentences *The system stopped the database*, *The system halted the database* and *The system terminated the database*, an SRL tool using PropBank tags '*the database*' with *A1*, indicating the database is the actor affected by the action. However, *A1* does not indicate that the three sentences have similar meanings (i.e., the verbs are synonyms). To identify similar sentences, UMTG employs semantic similarity detection techniques.

1. The term *phrase* indicates a word or a group of consecutive words.

TABLE 1
PropBank Additional Semantic Roles used in the paper.

| Verb-specific semantic roles | |
|---|---|
| **Identifier** | **Definition** |
| A0 | Usually indicates who performs an action. |
| A1 | Usually indicates the actor most directly affected by the action. |
| A2 | With motion verbs, indicates a final state or a location. |
| **Generic semantic roles** | |
| **Identifier** | **Definition** |
| AM-ADV | Adverbial modification. |
| AM-LOC | Indicates a location. |
| AM-MNR | Captures the manner in which an activity is performed. |
| AM-MOD | Indicates a modal verb. |
| AM-NEG | Indicates a negation, e.g. 'no'. |
| AM-TMP | Provides temporal information. |
| AM-PRD | Secondary predicate with additional information about A1. |

### 2.2 Semantic Similarity Detection

For semantic similarity detection, we use the VerbNet lexicon [59], which clusters verbs that *have a common semantics and share a common set of semantic roles* into a total of 326 verb classes [60]. Each verb class is provided with a set of *role patterns*. For example, $\langle A1,V\rangle$ and $\langle A0,V,A1\rangle$ are two *role patterns* for the VerbNet class *stop-55.4*, which includes, among others, the verbs *to stop*, *to halt* and *to terminate*. In $\langle A1,V\rangle$, the sentence contains only the verb ($V$), and the actor whose state is altered ($A1$). In $\langle A0,V, A1\rangle$, the sentence contains the actor performing the action ($A0$), the verb ($V$), and the actor affected by the action ($A1$). Examples of these two patterns are *the database stops* and *the system stops the database*, respectively. UMTG uses VerbNet version 3.2 [60], which includes 272 verb classes and 214 subclasses where a class may have more than one subclass.

VerbNet uses a model different than PropBank. There is a mapping between PropBank and the model in VerbNet [61]. For simplification, we use only PropBank role labels in the paper. All the verbs in a VerbNet class are guaranteed to have a common set of role patterns, but are not guaranteed to be synonyms (e.g., the verbs *repeat* and *halt* in the VerbNet class *stop-55.4*). We employ WordNet [62], a database of lexical relations, to cluster verbs with similar meaning. Further, we use WordNet to identify synonyms and antonyms of phrases in use case specifications (see Section 9.2).

### 2.3 Glossary

An *actor* specifies a type of role played by an entity interacting with a system (e.g., by exchanging signals and data), but which is external to the system (See Section 6).

A *use case* is a list of actions or event steps typically defining the interactions between an actor and a system to achieve a goal. It is generally named with a phrase that denotes the goal, e.g., *Identify Occupancy Status* (See Section 6).

A *use case specification* is a textual document that captures the specific details of a use case. Use case specifications provide a way to document the functional requirements of a system. They generally follow a template (See Section 6).

A *use case flow* is a possible sequence of interactions between actors and the system captured by a use case specification. A use case specification may include multiple alternative use case flows (See Section 6).

A *use case scenario* is a sequence of interactions between actors and the system. It represents a single use case execution. It is a possible path through a use case specification. It may include multiple use case flows (See Section 11).

An *abstract test case* is a human-readable description of the interactions between actors and the system under test that should be exercised during testing. It exercises one use case scenario. It includes one or more abstract test oracles (See Section 12).

An *abstract test oracle* is a human-readable description of a function that verifies if the behavior of the system meets its requirements. It captures the characteristics that the outputs generated by the system should have (See Section 12).

An *executable test case* is a sequence of executable instructions (i.e., invocations of test driver functions) that trigger the system under test, thus simulating the interactions between one or more actors and the system. It includes one or more executable test oracles (See Section 12).

An *executable test oracle* is an executable instruction that returns true when the systems behaves according to its requirements, false otherwise. An executable test oracle usually consists of a set of assertions verifying if a set of outputs generated by the system match a set of expected outputs (See Section 12).

A *test driver function* is a software module used to invoke the software under test. A test driver typically provides test inputs, controls and monitors execution, and reports test results [1]. See Section 12.

A *test verdict* (or *test result*) is an indication of whether or not an executed test case has passed or failed, i.e., if the actual output corresponds to the expected result or if deviations were observed [1]. A test verdict can be either *Pass* or *Fail*.

An *input equivalence partition* (also referenced as *equivalence partition* in this paper) is a subset of the range of values for an input variable, or set of input variables, for the software under test, such that all the values in the partition can reasonably be expected to be treated similarly by the software under test (i.e., they are considered equivalent) [1].

## 3 Motivation and Context

The context for which we developed UMTG is that of safety-critical embedded software in the automotive domain. The automotive domain is a representative example of the many domains for which compliance with requirements should be demonstrated through documented test cases. For instance, ISO-26262 [3], an automotive safety standard, states that all system requirements should be properly tested by corresponding system test cases.

In this paper, we use the system *BodySense*$^{TM}$ as one of the case studies and also to motivate and illustrate UMTG. *BodySense* is a safety-critical automotive software developed by IEE [21], a leading supplier of embedded software and hardware systems in the automotive domain. *BodySense* provides automatic airbag deactivation for child seats. It classifies vehicle occupants for smart airbag deployment. Using a capacitive sensor in the vehicle's passenger seat, it monitors whether the seat is occupied, as well as classifying

the occupant. If the passenger seat has a child in a child seat or is unoccupied, the system disables the airbag. For seats occupied by adult passengers, it ensures the airbag is deployed in the event of an accident. *BodySense* also provides occupant detection for the seat belt reminder function.

Table 2 gives a simplified version of a real test case for *BodySense*. Lines 1, 3, 5, 7, and 9 provide high-level operation descriptions, i.e., informal descriptions of the operations to be performed on the system. These lines are followed by the name of the functions that should be executed by the test driver along with the corresponding input and expected output values. For instance, Line 4 invokes the function *SetBus* with a value indicating that the test driver should simulate the presence of an adult on the seat (for simplicity assume that, when an adult is seated, the capacitance sensor positioned on a seat sends a value above 600 on the bus).

TABLE 2
An example test case for *BodySense*.

| Line | Operation | Inputs/Expectations |
|---|---|---|
| 1 | *Reset power and wait* | |
| 2 | ResetPower | Time=INIT_TIME |
| 3 | *Set occupant status - Adult* | |
| 4 | SetBus | Channel = RELAY<br>Capacitance = 601 |
| 5 | *Simulate a nominal temperature* | |
| 6 | SetBus | Channel=RELAY<br>Temperature = 20 |
| 7 | *Check that and Adult has been detected on the seat, i.e. SeatBeltReminder status is Occupied and AirBagControl status is Occupied.* | |
| 8 | ReadAndCheckBus | D0=OCCUPIED<br>D1=OCCUPIED |
| 9 | *Check that the AirBagControl has received new data.* | |
| 10 | CheckAirbagPin | 0x010 |

Exhaustive test cases needed to validate safety-critical, embedded software are difficult both to derive and maintain because requirements are often updated during the software lifecycle (e.g., when *BodySense* needs to be customized for new car models). For instance, the functional test suite for *BodySense* is made of 192 test cases which include a total of 4,707 calls to test driver functions and around 21,000 variable assignments. The effort required to specify test cases for *BodySense* is overwhelming. Without automated test case generation, such testing activity is not only expensive but also error prone.

Within the context of testing safety-critical, embedded software such as *BodySense*, we identify three challenges that need to be considered for the automatic generation of system-level, acceptance test cases from functional requirements:

***Challenge 1:*** **Feasible Modeling.** Most of the existing automatic system test generation approaches are model-based and rely upon behavioral models such as state, sequence or activity diagrams (e.g., [10], [63], [64], [65], [66]). In complex industrial systems, behavioral models that are precise enough to enable test automation are so complex that their specification cost is prohibitive and the task is often perceived as overwhelming by engineers. To evaluate the applicability of behavioral modeling on *BodySense*, we asked the IEE engineers to specify system sequence diagrams (SSDs) for some of the use cases of *BodySense*. For example, the SSD for the use case *Identify initial occupancy status of a seat* included 74 messages, 19 nested blocks, and 24 references to other SSDs that had to be derived. This was considered too complex for the engineers and required significant help from the authors of this paper, and many iterations and meetings. Our conclusion is that the adoption of behavioral modeling, at the level of detail required for automated testing, is not a practical option for acceptance testing automation unless detailed behavioral models are already used by engineers for other purposes, e.g., software design.

***Challenge 2:*** **Automated Generation of Test Data.** Without behavioral modeling, test generation can be driven only by existing requirements specifications in NL, which complicates the identification of the test data (e.g., the input values to send to the system under test). Because of this, most of the existing approaches focus on the identification of test scenarios (i.e., the sequence of activities to perform during testing), and ask engineers to manually produce the test data. Given the complexity of the test cases to be generated (recall that the BodySense test suite includes 21000 variable assignments), it is extremely important to automatically generate test data, and not just test scenarios.

***Challenge 3:*** **Deployment and Execution of the Test Suite.** Execution of test cases for a system like *BodySense* entails the deployment of software under test on the target environment. To speed up testing, test case execution is typically automated through test scripts invoking test driver functions. These functions simulate sensor values and read computed results from a communication bus. Any test generation approach should generate appropriate function calls and test data in a processable format for the test driver. For instance, the test drivers in *BodySense* need to invoke driver functions (e.g., *SetBus*) to simulate seat occupancy.

In the rest of this paper, we focus on how to best address these challenges in a practical manner, in the context of use case-driven development of embedded systems.

## 4 Related Work

In this section, we cover the related work across three categories in terms of the challenges we presented in Section 3.

***Feasible Modeling.*** Most of the system test case generation approaches require that system requirements be given in UML behavioral models such as activity diagrams (e.g., [10], [67], [68], [69]), statecharts (e.g., [11], [63], [70], [71]), and sequence diagrams (e.g., [5], [64], [72], [73]). For instance, Nebut et al. [5] propose a use case driven test generation approach based on system sequence diagrams. Gutierrez et al. [74] introduce a systematic process based on model-driven engineering paradigm to automate the generation of system cases from functional requirements given in activity diagrams. Briand and Labiche [64] use both activity and sequence diagrams to generate system test cases. While sequential dependencies between use cases are extracted from an activity diagram, sequences in a use case are derived from a system sequence diagram. In contrast, UMTG needs only use case specifications complemented by a domain model and OCL constraints. In addition, UMTG is

able to automatically generate most of the OCL constraints from use case specifications.

There are techniques generating behavioral models from NL requirements [12], [13], [14], [75], [76], [77]. Some approaches employ similar techniques in the context of test case generation. For instance, Frohlich and Link [78] generate test cases from UML statecharts that are automatically derived from use cases. De Santiago et al. [79] provide a similar approach to generate test cases from statecharts derived from NL scenario specifications. Riebisch et al. [80] describe a test case generation approach based on the semi-automated generation of state diagrams from use cases. Katara and Kervinen [81] propose an approach which generates test cases from labeled transition systems that are derived from use case specifications. Nogueira et al. [82] provide a test case generation approach using labelled transition systems derived from use case specifications. The formal testing theory the approach is built upon is proved to be sound: if a generated test case fails, it necessarily means that the system under test does not conform to the specification (according to a formally defined conformance relation). Sarmiento et al. [16], [17] propose another approach to generate test scenarios from a restricted form of NL requirements. The approach automatically translates restricted NL requirements into executable Petri-Net models; the generated Petri-Nets are used as input for test scenario generation. Soeken et al. [83] employ a statistical parser [84] and a lexical database [62] to semi-automatically generate sequence diagrams from NL scenarios, which are later used to semi-automatically generate test cases. Hartmann et al. [85] provide a test-generation tool that creates a set of test cases from UML models that are manually annotated and semi-automatically extracted from use case specifications. All these approaches mentioned above have two major drawbacks in terms of feasible modeling: (i) generated test sequences have to be edited, corrected, and/or refined and (ii) test data have to be manually provided in the generated test models. In contrast, UMTG not only generates sequences of function calls that do not need to be modified but also generates test data for function calls.

Kesserwan et al. [86] provide a model-driven testing methodology that supports test automation based on system requirements in NL. Using the methodology, the engineer first specifies system requirements according to Cockburn use case notation [87] and then manually refines them into Use Case Map (UCM) scenario models [88]. In addition, test input data need to be manually extracted from system requirements and modelled in a data model. UMTG requires that system requirements be specified in RUCM without any further refinement. Text2Test [47], [89] extracts control flow implicitly described in use case specifications, which can be used to automatically generate system test cases. The adaptation of such an approach in the context of test case generation has not been investigated.

***Automated Generation of Test Data.*** The ability to generate test data, and not just abstract test scenarios, is an integral part of automated test case generation [90]. However, many existing NL-based test case generation approaches require manual intervention to derive test data for executable test cases (e.g., [15], [16], [17], [86]), while some other approaches focus only on generating test data (e.g., [91], [92], [93], [94], [95], [96]). For instance, Zhang et al. [15] generate test cases from RUCM use cases. The generated test cases cannot be executed automatically because they do not include test data. Sarmiento et al. [16] generate test scenarios without test data from a restricted form of NL requirements specifications.

Similar to UMTG, Kaplan et al. [97] propose another approach, i.e., Archetest, which generates test sequences and test inputs from a domain model and use case specifications together with invariants, guardconditions and postconditions. Yue et al. [24] propose a test case generation tool (aToucan4Test), which takes RUCM use case specifications annotated with OCL constraints as input and generates automatically executable test cases. These two test generation approaches require that conditions and constraints be provided by engineers to automatically generate test data. In contrast, UMTG can automatically generate, from use case specifications, most of the OCL constraits that are needed for the automated generation of test data.

In some contexts, test data might be simple and consist of sequences of system events without any associated additional parameter value. This is the case of interaction test cases for smartphone systems, which can be automatically generated by the approach proposed by De Figueiredo et al. [19]. The approach processes use case specifications in a custom use case format to derive sequences of system operations and events. UMTG complements this approach with the generation of parameter values, which is instead needed to perform functional testing at the system level.

Carvalho et al. [20], [98] generate executable test cases for reactive systems from requirements written according to a restricted grammar and dictionary. The proposed approach effectively generates test data but has two main limitations: (i) the underlying dictionary may change from project to project (e.g., the current version supports only seven verbs of the English language), and (ii) the restricted grammar may not be suitable to express some system requirements (e.g., the approach does not tackle the problem of processing transitive and intransitive forms of the same verb). In contrast, UMTG does not impose any restricted dictionary or grammar but simply relies on a use case format, RUCM, which can be used to express use cases for different kinds of systems. RUCM does not restrict the use of verbs or nouns in use case steps and thus does not limit the expressiveness of use case specifications. Furthermore, the RUCM keywords are used to specify input and output steps but do not constraint internal steps or condition sentences (see Section 6). Finally, by relying on SRL and VerbNet, UMTG aims to ensure the correct generation of OCL constraints (see Section 9), without restricting the writing of sentences (e.g., it supports the use of both transitive and intransitive forms).

Other approaches focus on the generation of class invariants and method pre/postconditions, from NL requirements, which, in principle, could be used for test data generation (e.g., [48], [49], [99]). Pandita et al. [99] focus only on API descriptions written according to a CNL. NL2OCL [48] and NL2Alloy [49], instead, process a UML class diagram and NL requirements to derive class invariants and method pre/postconditions. These two approaches rely on an ad-hoc semantic analysis algorithm that uses information in

the UML class diagram (e.g., class and attribute names) to identify the roles of words in sentences. They rely on the presence of specific keywords to determine passive voices and to identify the operators to be used in the generated invariants and conditions. Their constraint generation is rule-based, but they do not provide a solution to ease the processing of a large number of verbs with a reasonable number of rules. Thanks to the use of Wordnet synsets and VerbNet classes (see Section 9), UMTG can process a large set of verbs with few rules to generate OCL constraints.

Though NL2OCL [48] and NL2Alloy [49] are no longer available for comparison, they seem more useful for deriving class invariants including simple comparison operators (i.e., the focus of the evaluation in [48]), rather than for generating pre/postconditions of the actions performed by the system (i.e., the focus of UMTG). Pre/postconditions are necessary for deriving test data in our context.

***Deployment and Execution of the Test Suite.*** The generation of executable test cases impacts on the usability of test generation techniques. In code-based approaches (e.g., [100], [101]), the generation of executable test cases is facilitated by the fact that it is based on processing the interfaces used during the test execution (e.g., test driver API).

In model-based testing, the artefacts used to drive test generation are software abstractions (e.g., UML models). In this context, the generation of executable test cases is usually based on adaptation and transformation approaches [44]. The adaptation approaches require the implementation of a software layer that, at runtime, matches high-level operations to software interfaces. They support the execution of complex system interactions (e.g., they enable feedback-driven, model-based test input generation [102]). The transformation approaches, instead, translate an abstract test case into an executable test case by using a mapping table containing regular expressions for the translation process. They require only abstract test cases and a mapping table, while the adaptation approaches need communication channels between the software under test and the adaptation layer, which might not be possible for many embedded systems. Therefore, UMTG uses a mapping table that matches abstract test inputs to test driver function calls.

Model Transformation by Example (MTBE) approaches aim to learn transformation programs from source and target model pairs supplied as examples (e.g., [103], [104], [105]). These approaches search for a model transformation in a space whose boundaries are defined by a model transformation language and the source and target metamodels [106]. Given the metamodels of abstract and executable test cases, MTBE can be applied to automatically generate part of the mapping table as a transformation program. However, this solution can be considered only when there are already some example abstract and executable test cases, which is not the case in our context, and we leave it for future work.

In Table 3, based on a set of features necessary for the automatic generation of system test cases for acceptance testing, we summarize the differences between UMTG and prominent related work. For each approach, the symbol '+' indicates that the approach provides the feature, the symbol '-' indicates that it does not provide the feature, and 'NA' indicates that the feature is not applicable because it is out of its scope. For instance, the approach by Yue et al. [76] automatically generates UML analysis models, not test cases. Therefore, all the features related to the generation of test input sequences and test data are not considered for Yue et al. [76] in Table 3. Most of the existing approaches extract behavioral models from NL specifications [75], [76], [77], generate abstract test inputs sequences [5], [15], [16], [17], [64], [74], [78], [79], [81], [83], [86] or derive test input data [91], [92], [93], [94], [95], [96]. The few approaches generating both abstract test input sequences and test data [20], [24], [97], [98] either require the specification of data constraints using a formal language [24], [97] or rely on a restricted grammar or dictionary [20], [98]. Finally, a few approaches focus only on the automated generation of data constraints from text in NL [48], [49], [99]. UMTG is the only approach that automates the generation of both test input sequences and test input data, without requiring neither behavioral models nor a very restricted language. In the latter case, CNLs are limited to a restricted vocabulary while UMTG only requires the use of simple sentences and keywords but otherwise allows the use of the full English vocabulary. Test input generation is enabled by the capability of automatically extracting data constraints from NL, a feature that has not so far been integrated by existing techniques.

## 5 Overview of the Approach

The process in Fig. 1 presents an overview of our approach. In UMTG, behavioral information and high-level operation descriptions are extracted from use case specifications (*Challenge 1*). UMTG generates OCL constraints from the use case specifications, while test inputs are generated from the OCL constraints through constraint solving (*Challenge 2*). Test driver functions corresponding to the high-level operation descriptions and oracles implementing the postconditions in the use case specifications are generated through the mapping tables provided by the engineer (*Challenge 3*).

The engineer elicits requirements with RUCM (Step 1). The domain model is manually created as a UML class diagram (Step 2). UMTG automatically checks if the domain model includes all the entities mentioned in the use cases (Step 3). NLP is used to extract domain entities from the use cases. Missing entities are shown to the engineer who refines the domain model (Step 4). Steps 3 and 4 are iterative: the domain model is refined until it is complete.

Once the domain model is complete, most of the OCL constraints are automatically generated from the extracted conditions (Step 5). The generated constraints are manually inspected by engineers to ensure they capture the meaning of the use case conditions and, as a result, guarantee the soundness of the generated test cases (i.e., to ensure that correct implementations are never rejected). Based on experience in different testing contexts, we conjecture that the manual validation of constraints is significantly less expensive than their manual definitions [107], [108].

The engineer manually writes the few OCL constraints that cannot be automatically generated (Step 6). UMTG further processes the use cases with the OCL constraints to generate a use case test model for each use case specification (Step 7). A use case test model is a directed graph that

TABLE 3
Summary and comparison of the related work.

| | No need for behavioral models | Automated extraction of control flow from NL text | No need for restricted grammar or dictionary | Automated generation of abstract test input sequences | No need for editing the generated test input sequences | Automated generation of test data | No need for formal data constraints | Automated generation of data constraints from NL text |
|---|---|---|---|---|---|---|---|---|
| **UMTG** | + | + | + | + | + | + | + | + |
| Nebut et al. [5] | − | − | *NA* | + | − | − | *NA* | *NA* |
| Gutierrez et al. [74] | − | − | *NA* | + | − | − | *NA* | *NA* |
| Briand et al. [64] | − | − | *NA* | + | − | − | *NA* | *NA* |
| Yue et al. [76] | + | + | + | *NA* | *NA* | *NA* | *NA* | *NA* |
| Gutierrez et al. [75] | + | + | + | *NA* | *NA* | *NA* | *NA* | *NA* |
| Ding et al. [77] | + | + | + | *NA* | *NA* | *NA* | *NA* | *NA* |
| Frohlich and Link [78] | + | + | + | + | − | − | *NA* | *NA* |
| De Santiago et al. [79] | + | + | + | + | − | − | *NA* | *NA* |
| Katara et al. [81] | + | + | − | + | − | − | *NA* | *NA* |
| Sarmiento et al. [16], [17] | + | + | − | + | − | − | *NA* | *NA* |
| Soeken et al. [83] | + | + | + | + | − | − | *NA* | *NA* |
| Kesserwan et al. [86] | − | − | *NA* | + | − | − | *NA* | *NA* |
| Text2Test [47], [89] | + | + | + | *NA* | *NA* | *NA* | *NA* | *NA* |
| Zhang et al. [15] | + | + | + | + | − | − | *NA* | *NA* |
| Weyuker et al. [91] | *NA* | *NA* | *NA* | *NA* | *NA* | + | − | − |
| Offutt et al. [92] | *NA* | *NA* | *NA* | *NA* | *NA* | + | − | − |
| Offutt et al. [93] | *NA* | *NA* | *NA* | *NA* | *NA* | + | − | − |
| Benattou et al. [94] | *NA* | *NA* | *NA* | *NA* | *NA* | + | − | − |
| Soltana et al. [95] | *NA* | *NA* | *NA* | *NA* | *NA* | + | − | − |
| Ali et al. [96] | *NA* | *NA* | *NA* | *NA* | *NA* | + | − | − |
| Carvalho et al. [20], [98] | + | + | − | + | + | + | + | + |
| Kaplan et al. [97] | + | + | + | + | + | + | − | − |
| Yue et al. [24] | + | + | + | + | + | + | − | − |
| Pandita et al. [99] | *NA* | *NA* | *NA* | *NA* | *NA* | *NA* | *NA* | + |
| NL2OCL [48] | *NA* | *NA* | *NA* | *NA* | *NA* | *NA* | *NA* | + |
| NL2Alloy [49] | *NA* | *NA* | *NA* | *NA* | *NA* | *NA* | *NA* | + |

explicitly captures the implicit behavioral information in a use case specification.

UMTG employs constraint solving for OCL constraints to generate test inputs associated with use case scenarios (Step 8). We use the term use case scenario for a sequence of use case steps that starts with a use case precondition and ends with a postcondition of either a basic or alternative flow. Test inputs cover all the paths in the testing model, and therefore all possible use case scenarios.

The engineer provides a mapping table that maps high-level operation descriptions and test inputs to the concrete driver functions and inputs that should be executed by the test cases (Step 9). Executable test cases are automatically generated through the mapping table (Step 10). If the test infrastructure and hardware drivers change in the course of the system lifespan, then only this table needs to change.

The rest of the paper explains the details of each step in Fig. 1, with a focus on how we achieved our automation objectives.

## 6 Elicitation of Requirements

Our approach starts with the elicitation of requirements in RUCM (Step 1 in Fig. 1). RUCM has a template with keywords and restriction rules to reduce ambiguity in use case specifications [12]. Since it was not originally designed for test generation, we introduce some extensions to RUCM.

Table 4 provides a simplified version of three *BodySense* use case specifications in RUCM (i.e., *Identify Occupancy Status*, *Self Diagnosis*, and *Classify Occupancy Status*). We omit some basic information such as actors and dependencies.

The use cases contain basic and alternative flows. A basic flow describes a main successful scenario that satisfies stakeholder interests. It contains a sequence of steps and a postcondition (Lines 5-11). A step can describe one of the following activities: an actor sends data to the system (Lines 5 and 41); the system validates some data (Line 7); the system replies to an actor with a result (Line 9); the system alters its internal state (Line 15). The inclusion of another use case is specified as a step using the keyword *INCLUDE USE CASE* (Line 6). All keywords are written in capital letters for readability.

The keyword *VALIDATES THAT* indicates a condition that must be true to take the next step; otherwise an alternative flow is taken. For example, the condition in Line 7 indicates that no error condition should be detected or qualified to proceed with the step in Line 8 and, otherwise, the alternative flow in Line 20 is taken. In *BodySense*, an error is qualified (i.e., confirmed), when it remains detected for 3400 ms.

Alternative flows describe other scenarios than the main one, both success and failure. An alternative flow always depends on a condition. In RUCM, there are three types of alternative flows: *specific*, *bounded* and *global*. For specific and bounded alternative flows, the keyword *RFS* is used to refer to one or more reference flow steps (e.g., Lines 21 and 13). A specific alternative flow refers to a step in its reference flow (Line 21). A bounded alternative flow refers to more

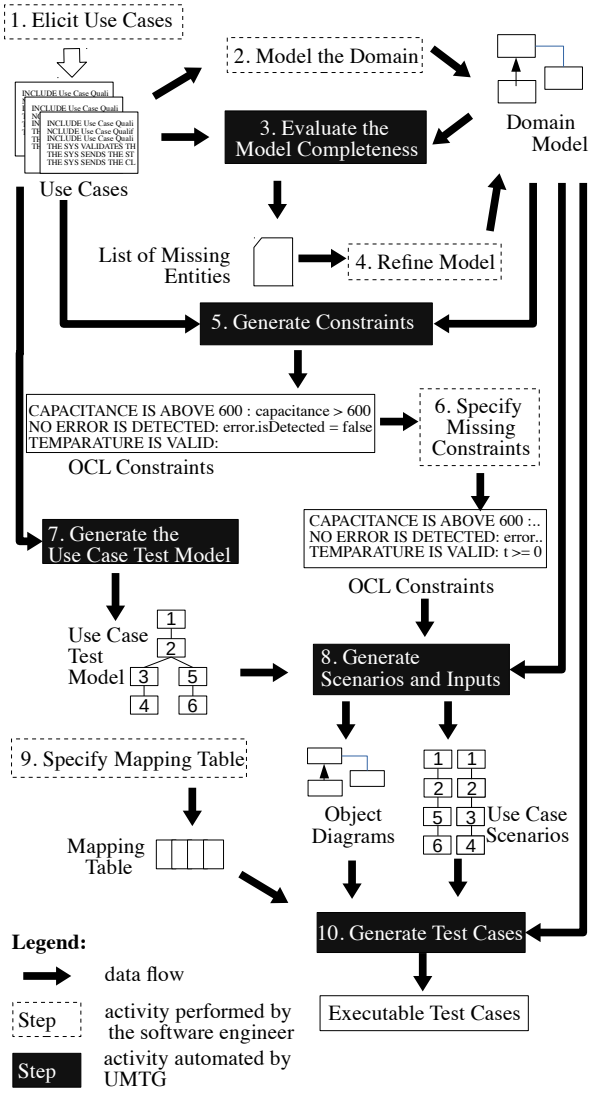


Fig. 1. Overview of the UMTG approach.

TABLE 4
Part of *BodySense* Use Case Specifications

| | |
|---|---|
| 1 | **1. Use Case** Identify Occupancy Status |
| 2 | **1.1 Precondition** |
| 3 | The system has been initialized. |
| 4 | **1.2 Basic Flow** |
| 5 | 1. The SeatSensor SENDS capacitance TO the system. |
| 6 | 2. INCLUDE USE CASE Self Diagnosis. |
| 7 | 3. The system VALIDATES THAT no error is detected and no error is qualified. |
| 8 | 4. INCLUDE USE CASE Classify Occupancy Status. |
| 9 | 5. The system SENDS the occupant class for airbag control TO AirbagControlUnit. |
| 10 | 6. The system SENDS the occupant class for seat belt reminder TO SeatBeltControlUnit. |
| 11 | Postcondition: The occupant class for airbag control has been sent to AirbagControlUnit. The occupant class for seat belt reminder has been sent to SeatBeltControlUnit. |
| 12 | **1.3 Bounded Alternative Flow** |
| 13 | RFS 2-4 |
| 14 | 1. IF voltage error is detected THEN |
| 15 | 2. The system resets the occupant class for airbag control to error. |
| 16 | 3. The system resets the occupant class for seat belt reminder to error. |
| 17 | 4. ABORT |
| 18 | 5. ENDIF |
| 19 | Postcondition: The occupant classes have been reset to error. |
| 20 | **1.4 Specific Alternative Flow** |
| 21 | RFS 3 |
| 22 | 1. IF some error has been qualified THEN |
| 23 | 2. The system SENDS the error occupant class TO AirbagControlUnit. |
| 24 | 3. The system SENDS the error occupant class TO SeatBeltControlUnit. |
| 25 | 4. ABORT |
| 26 | 5. ENDIF |
| 27 | Postcondition: The error occupant class has been sent to AirbagControlUnit. The error occupant class has been sent to SeatBeltControlUnit. |
| 28 | **1.5 Specific Alternative Flow** |
| 29 | RFS 3 |
| 30 | 1. The system SENDS the previous occupant class for airbag control TO AirbagControlUnit. |
| 31 | 2. The system SENDS the previous occupant class for seat belt reminder TO SeatBeltControlUnit. |
| 32 | 3. ABORT |
| 33 | Postcondition: The previous occupant class for airbag control has been sent to AirbagControlUnit. The previous occupant class for seat bet reminder has been set to SeatBeltControlUnit. |
| 34 | **2. Use Case** Self Diagnosis |
| 35 | **2.1 Precondition** |
| 36 | The system has been initialized. |
| 37 | **2.2 Basic Flow** |
| 38 | 1. The system sets temperature errors to not detected. |
| 39 | 2. The system sets memory errors to not detected. |
| 40 | 3. The system VALIDATES THAT the NVM is accessible. |
| 41 | 4. The system REQUESTS the temperature FROM the SeatSensor. |
| 42 | 5. The system VALIDATES THAT the temperature is above -10 degrees. |
| 43 | 6. The system VALIDATES THAT the temperature is below 50 degrees. |
| 44 | 7. The system sets self diagnosis as completed. |
| 45 | Postcondition: Error conditions have been examined. |
| 46 | **2.3 Specific Alternative Flow** |
| 47 | RFS 3 |
| 48 | 1. The System sets MemoryError to detected. |
| 49 | 2. RESUME STEP 4 |
| 50 | Postcondition: The system has detected a MemoryError. |
| 51 | **2.4 Specific Alternative Flow** |
| 52 | RFS 5 |
| 53 | 1. The System sets TemperatureLowError to detected. |
| 54 | 2. RESUME STEP 7 |
| 55 | Postcondition: The system has detected a TemperatureLowError. |
| 56 | **2.5 Specific Alternative Flow** |
| 57 | RFS 6 |
| 58 | 1. The System sets TemperatureHighError to detected. |
| 59 | 2. RESUME STEP 7 |
| 60 | Postcondition: The system has detected a TemperatureHighError. |
| 61 | **3. Use Case** Classify Occupancy Status |
| 62 | **3.1 Precondition** |
| 63 | The system has been initialized. |
| 64 | **3.2 Basic Flow** |
| 65 | 1. The system sets the occupant class for airbag control to Init. |
| 66 | 2. The system sets the occupant class for seatbelt reminder to Init. |
| 67 | 4. The system VALIDATES THAT the capacitance is above 600. |
| 68 | 5. The system sets the occupant class for airbag control to Occupied. |
| 69 | 6. The system sets the occupant class for seatbelt reminder to Occupied. |
| 70 | Postcondition: An adult has been detected on the seat. |
| 71 | **3.3 Specific Alternative Flow** |
| 72 | RFS 4 |
| 73 | 1. IF capacitance is above 200 THEN |
| 74 | 2. The system sets the occupant class for airbag control to Empty. |
| 75 | 3. The system sets the occupant class for seatbelt reminder to Occupied. |
| 76 | 4. EXIT |
| 77 | 5. ENDIF |
| 78 | Postcondition: A child has been detected on the seat. |
| 79 | **3.4 Specific Alternative Flow** |
| 80 | RFS 4 |
| 81 | 1. The system sets the occupant class for airbag control to Empty. |
| 82 | 2. The system sets the occupant class for seatbelt reminder to Empty. |
| 83 | 3. EXIT |
| 84 | Postcondition: The seat has been recognized as being empty. |

than one step in the reference flow (Line 13) while a global alternative flow refers to any step in the reference flow.

Bounded and global alternative flows begin with the keyword *IF .. THEN* for the condition under which the alternative flow is taken (Line 14). Specific alternative flows do not necessarily begin with *IF .. THEN* since a condition may already be indicated in its reference flow step (Line 7). In Line 71, we have an example of a specific alternative flow beginning with an *IF .. THEN* condition. The alternative flows are evaluated in the order they appear in the specification.

UMTG introduces extensions into RUCM regarding the IF conditions, the keyword EXIT, and the way input/output messages are expressed UMTG prevents the definition of use case flows containing branches [22], thus enforcing the adoption of *IF* conditions only as a means to specify guard conditions for alternative flows. UMTG introduces the keywords *SENDS ... TO* and *REQUESTS ... FROM* for the system-actor interactions. Depending on the subject of the sentence, the former indicates either that an actor provides an input to the system (Line 5) or that the system provides an output to an actor (Line 9). The latter is used only for inputs, and indicates that the input provided by the actor has

been requested by the system (Line 41). UMTG introduces the keyword *EXIT* to indicate use case termination under alternative valid execution conditions (Line 76 describing the case of a child being detected on a seat). The keyword *EXIT* complements the keyword *ABORT*, which is used to indicate the abnormal use case termination (Line 17).

## 7 Tagging of Use Case Specifications

We implemented an NLP application to extract the information required for three UMTG steps in Fig. 1: *evaluate the model completeness* (Step 3 in Fig. 1), *generate OCL constraints* (Step 5), and *generate the use case test model* (Step 7). This application annotates the phrases in the use case specification sentences to enable the generation of use case models and to identify the steps for which an OCL constraint should be generated. It does not include the procedures for generating OCL constraints (i.e., SRL and semantic similarity detection) because these are integrated in a dedicated algorithm described in Section 9.

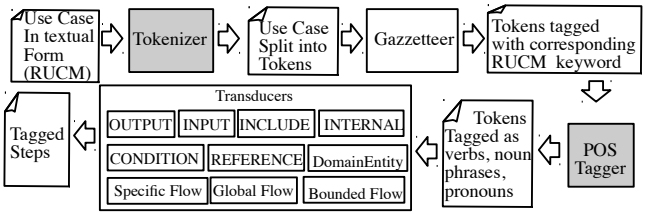


Fig. 2. NLP pipeline applied to extract data used by UMTG Steps 3, 5, and 7.

The NLP application is based on the GATE workbench [109], an open source NLP framework, and implements the NLP pipeline in Fig. 2. The pipeline includes both default NLP components (grey) and components built to process use case specifications in RUCM (white). The *Tokenizer* splits the use cases into tokens. The *Gazetteer* identifies the RUCM keywords. For example, according to RUCM, the system under test is expected to be referred to with the keyword *The system*. The *POS Tagger* tags tokens according to their nature: *verb*, *noun*, and *pronoun*. The pipeline is terminated by a set of *transducers* that tag blocks of words with additional information required by the three UMTG steps. The transducers integrated in UMTG (1) identify the kinds of RUCM steps (i.e., output, input, include, condition and internal steps), (2) distinguish alternative flows, and (3) detect RUCM references (i.e., the RFS keyword), conditions, and domain entities in the use case steps.

Fig. 3 gives an example transducer for condition steps. The arrow labels in higher case represent the transducer's inputs, i.e., tags previously identified by the POS tagger, the gazzetteer or other transducers. The italic labels show the tags assigned by the transducer to the words representing the transducer's input. Fig. 4 gives the tags associated with the use case step in Line 67 of Table 4 after the execution of the transducer in Fig. 3. In Fig. 4, multiple tags are assigned to the same blocks of words. For example, the noun phrase '*the capacitance*' is tagged both as a *domain entity* and as part of a *condition*.

The tags generated by transducers are used in Steps 3, 5, and 7 of Fig. 1. Table 5 provides, for each generated tag, the

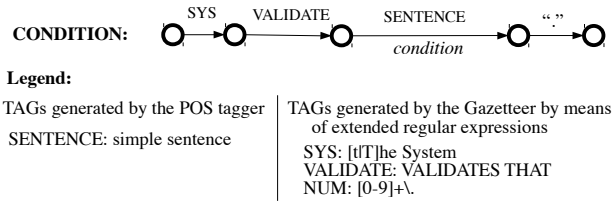


Fig. 3. Part of the transducer that identifies conditions.

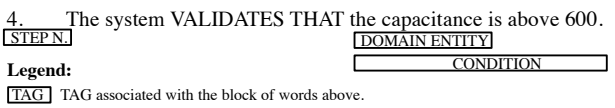


Fig. 4. Tags associated with the use case step in Line 67 of Table 4.

list of UMTG steps that process the associated phrases. For example, the noun phrases annotated with the tag *domain entity* are used in Step 3 to determine if the domain model is complete and, in Step 7, to identify inputs. Further details are provided in the following sections.

## 8 Evaluation of the Domain Model Completeness

The completeness of the domain model is important to generate correct and complete test inputs. UMTG automatically identifies missing domain entities to evaluate the model completeness (Step 3 in Fig. 1). This is done by checking correspondences between the domain model elements and the domain entities identified by the NLP application.

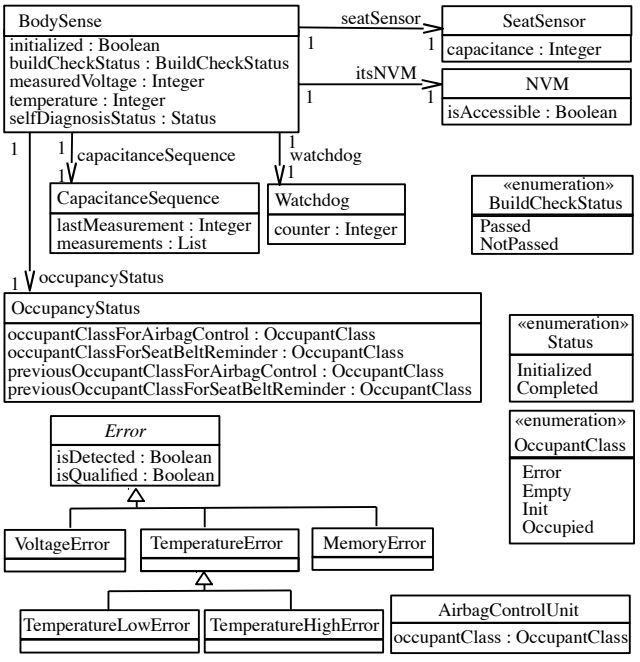


Fig. 5. Part of the domain model for $BodySense$.

Domain entities in a use case may not be modelled as classes but as attributes. Fig. 5 shows a simplified excerpt of the domain model for *BodySense* where the domain entities

TABLE 5
List of UMTG steps in Fig 1 that process the tags generated by transducers.

| Transducer Tag | UMTG Step(s) |
|---|---|
| OUTPUT STEP | Step 7. |
| INPUT STEP | Step 7. |
| INCLUDE STEP | Step 7. |
| CONDITION STEP | Step 5, Step 7. |
| INTERNAL STEP | Step 7. |
| BASIC FLOW | Step 7. |
| ALTERNATIVE FLOW | Step 7. |
| RFS | Step 7. |
| CONDITION | Step 5. |
| DOMAIN ENTITY | Step 3, Step 7. |

'*occupant class for airbag control*' and '*occupant class for seat belt reminder*' are modelled as attributes of the class *OccupancyStatus*. UMTG follows a simple yet effective solution to check entity and attribute names. For each domain entity identified through NLP, UMTG generates an entity name by removing all white spaces and by putting all first letters following white spaces in capital. For instance, the domain entity '*occupant class for airbag control*' becomes '*OccupantClassForAirbagControl*'. UMTG checks the string similarity between the generated entity names and the domain model elements. Engineers are asked to correct their domain model and use case specifications in the presence of confirmed mismatches.

## 9 Generation of OCL Constraints

To identify test inputs via constraint solving, UMTG needs to derive OCL constraints that capture (1) the effect that the internal steps have on the system state (i.e., the postconditions of the internal steps), (2) the use case preconditions, and (3) the conditions in the condition steps. For instance, for the basic flow of the use case *Classify Occupancy Status* (Lines 64 to 70 in Table 4), we need a test input that satisfies the condition *'the capacitance is above 600'* (Line 67).

As part of UMTG, we automate the generation of OCL constraints (Step 5 in Fig. 1). Using some predefined constraint generation rules (hereafter transformation rules), UMTG automatically generates an OCL constraint for each precondition, internal step and condition step identified by the transducers in Fig. 2. The generated constraint captures the meaning of the NL sentence in terms of the concepts in the domain model. Table 6 shows some of the OCL constraints generated from the use case specifications in our case studies in Section 14.

Section 9.1 summarizes our assumptions for the generation of OCL constraints. Section 9.2 describes the constraint generation algorithm. In Section 9.3, we discuss the correctness and generalizability of the constraint generation.

### 9.1 Working Assumptions

The constraint generation is enabled by three assumptions.

*Assumption 1 (Domain Modeling).* There are domain modeling practices common for embedded systems:

A1.1 Most of the entities in the use case specifications are given as classes in the domain model.

A1.2 The names of the attributes and associations in the domain model are usually similar with the phrases in the use case specifications.

A1.3 The attributes of domain entities (e.g., *Watchdog.counter* in Fig. 5) are often specified by possessive phrases (i.e., genitives and of-phrases such as *of the watchdog* in S12 in Table 6) and attributive phrases (e.g., *watchdog* in S13) in the use case specifications.

A1.4 The domain model often includes a system class with attributes that capture the system state (e.g., *BodySense* in Fig. 5).

A1.5 Additional domain model classes are introduced to group concepts that are modelled using attributes.

A1.6 Discrete states of domain entities are often captured using either boolean attributes (e.g., *isAccessible* in Fig. 5), or attributes of enumeration types (e.g., *BuildCheckStatus::Passed* in Fig. 5).

To ensure that Assumption 1 holds, UMTG iteratively asks engineers to correct their models (see Section 8). With Assumption 1, we can rely on string syntactic similarity to select the terms in the OCL constraints (i.e., classes and attributes in the domain model) based on the phrases appearing in the use case steps. String similarity also allows for some degree of flexibility in naming conventions.

*Assumption 2 (OCL constraint pattern).* The conditions in the use case specifications of embedded systems are typically simple and capture information about the state of one or more domain entities (i.e., classes in the domain model). For instance, in *BodySense*, the preconditions and condition steps describe safety checks ensuring that the environment has been properly set up (e.g., S3 in Table 6), or that the system input has been properly obtained (e.g., S5), while the internal steps describe updates on the system state (e.g., S2). They can be expressed in OCL using the pattern in Fig. 6, which captures assignments, equalities, and inequalities.

The generated constraints include an entity name (`ENTITY` in Fig. 6), an optional selection part (`SELECTION`), and a query element (`QUERY`). The query element can be specified according to three distinct sub-patterns: `FORALL`, `EXISTS` and `COUNT`. `FORALL` specifies that a certain expression (i.e., `EXPRESSION`) should hold for all the instances `i` of the given entity; `EXISTS` indicates that the expression should hold for at least one of the instances. `COUNT` is used when the expression should hold for a certain number of instances. Examples of these three query elements are given in the OCL constraints generated for the sentences S4, S10, and S11 in Table 6, respectively. The pattern `EXPRESSION` contains a left-hand side variable (hereafter *lhs-variable*), an OCL operator, and a right-hand side term (hereafter *rhs-term*), which is either another variable or a literal. The lhs-variable indicates an attribute of the entity whose state is captured by the constraint, while the rhs-term captures the state information (e.g., the value expected to be assigned to an attribute). The optional selection part selects a subset of all the available instances of the given entity type based on their subtype; an example is given in the OCL constraint for S6 in Table 6.

*Assumption 3 (SRL).* The SRL toolset (the CNP tool in our implementation) identifies all the semantic roles in a

TABLE 6
Some constraints from the BodySense and *HOD* case studies in Section 14, with tags generated by SRL.

| # | Sentence with SRL tags | Corresponding OCL Constraint |
|---|---|---|
| S1 | {The system VALIDATES THAT}$_{ignored}$ {the capacitance}$_{A1}$ {is}$_{verb}$ {above 600}$_{AM-LOC}$ | $BodySense.allInstances() \rightarrow forAll(b\|b.seatsensor.capacitance > 600)$ |
| S2 | {The system}$_{A0}$ {sets}$_{verb}$ {the occupant class for airbag control}$_{A1}$ {to Init}$_{A2}$ | $BodySense.allInstances() \rightarrow forAll(b\|b.occupancyStatus.occupant ClassForAirbagControl = OccupantClass :: Init)$ |
| S3 | {The system VALIDATES THAT}$_{ignored}$ {the NVM}$_{A1}$ {is}$_{verb}$ {accessible}$_{AM-PRD}$ | $BodySense.allInstances() \rightarrow forAll(i\|i.itsNVM.isAccessible = true)$ |
| S4 | {The system}$_{A0}$ {sets}$_{verb}$ {temperature errors}$_{A1}$ {to detected}$_{A2}$ | $TemperatureError.allInstances() \rightarrow forAll(i\|i.isDetected = true)$ |
| S5 | {The system VALIDATES THAT}$_{ignored}$ {the build check}$_{A1}$ {has been passed}$_{verb}$ | $BodySense.allInstances() \rightarrow forAll(i\|i.buildCheckStatus = BuildCheckStatus :: Passed)$ |
| S6 | {The system VALIDATES THAT}$_{ignored}$ {no}$_{NOM-NEG}$ {error [(except voltage errors)$_{NOM_{NP}}$ and (memory errors)$_{NOM_{NP}}$]$_{NOM-A2}$ }$_{A1}$ {is detected}$_{verb}$ | $Errors.allInstances() \rightarrow select(e\|\ not\ e.typeOf(VoltageError) and\ not\ e.typeOf(MemoryError)) \rightarrow forAll(i\|i.isDetected = false)$ |
| S7 | {The system}$_{A0}$ {erases}$_{verb}$ {the measured voltage}$_{A1}$ | $BodySense.allInstances() \rightarrow forAll(i\|i.measuredVoltage = 0)$ |
| S8 | {The system}$_{A0}$ {erases}$_{verb}$ {the occupant class}$_{A1}$ {from the airbag control unit}$_{A2}$ | $AirbagControlUnit.allInstances() \rightarrow forAll(i\|i.occupantClass = null)$ |
| S9 | {The system}$_{A0}$ {disqualifies}$_{verb}$ {temperature errors}$_{A1}$ | $TemperatureError.allInstances() \rightarrow forAll(i\|i.isQualified! = true)$ |
| S10 | {IF}$_{ignored}$ {some error}$_{A1}$ {has been qualified}$_{verb}$. | $Error.allInstances() \rightarrow exists(i\|i.isQualified = true)$ |
| S11 | {The system VALIDATES THAT}$_{ignored}$ {the driver}$_{A1}$ {put}$_{verb}$ {two hands}$_{A1}$ {on the steering wheel}$_{AM-LOC}$. | $Hand.allInstances() \rightarrow select(i\|i.onTheSteeringWheel = true) \rightarrow size() = 2$ |
| S12 | {The system}$_{A0}$ {resets}$_{verb}$ {the counter of the watchdog}$_{A1}$ | $BodySense.allInstances() \rightarrow forAll(i\|i.watchdog.counter = 0)$ |
| S13 | {The system}$_{A0}$ {resets}$_{verb}$ {the watchdog counter}$_{A1}$ | $BodySense.allInstances() \rightarrow forAll(i\|i.watchdog.counter = 0)$ |
| S14 | {The system VALIDATES THAT}$_{ignored}$ {the last measurement of the capacitance sequence}$_{A1}$ {is}$_{verb}$ {above 40}$_{AM-LOC}$ | $BodySense.allInstances() \rightarrow forAll(i\|i.capacitanceSequence .lastMeasurement = 0)$ |

**Notes.** Sentences S1, S2, S8, S14 are taken from the *BodySense* case study system, sentence S11 from *HOD*; the other sentences are shared by both. In S6, different types of brackets are used to annotate phrases with nested semantic roles.

**CONSTRAINT** = **ENTITY** .allInstances() **[SELECTION] QUERY**
**QUERY** = **FORALL** | **EXISTS** | **COUNT**
**FORALL** = → forAll ( **EXPRESSION** )
**EXISTS** = → exists ( **EXPRESSION** )
**COUNT** = → select ( **EXPRESSION** ) → size() **OPERATOR NUMBER**
**EXPRESSION** = i | i.**LHS-VARIABLE OPERATOR RHS-TERM**
**LHS-VARIABLE** = **VARIABLE**
**RHS-TERM** = **VARIABLE** | **LITERAL**
**VARIABLE** = **ATTR** | **ASSOC** { . **ATTR** | **ASSOC** }
**SELECTION** = → select( e | **TYPESEL** { and **TYPESEL** } )
**TYPESEL** = not e.TypeOf( **CLASS** )

**Note:** This pattern is expressed using a simplified EBNF grammar [110] where non-terminals are bold and terminals are not bold. **ENTITY** stands for a class in the domain model. **LITERAL** is an OCL literal (e.g., '1' or 'a'). **NUMBER** is an OCL numeric literal (e.g., '1'). **ATTR** is an attribute of a class in the domain model. **ASSOC** is an association end in the domain model. **OPERATOR** is a math operator ( -, +, =, <, ≤, ≥, >).

Fig. 6. Pattern of the OCL constraints generated by UMTG.

**Require:** $S_{NL}$, a sentence in natural language
**Require:** $domainModel$, a domain model
**Ensure:** $\langle ocl, score \rangle$, an OCL constraint with a score
1 **function** GENERATEOCL($S_{NL}$, domainModel)
2 $S_{srl}$ ←generate a sentence annotated with SRL roles from $S_{NL}$
3 $transformationRules$ ←identify the transformation rules to apply
4 based on the verb in $S_{srl}$
5 **for each** $transformationRule$ **do**
6 generate a new OCL constraint ($constraint$) by applying
7 the $transformationRule$
8 $constraints \leftarrow constraints \cup constraint$
9 **end for**
10 **return** the ocl constraint with the best score
11 **end function**

Fig. 7. The OCL constraint generation algorithm.

sentence that are needed to correctly generate an OCL constraint. Our transformation rules use the roles to correctly select the domain model elements to be used in the OCL constraint (see Section 9.2).

Table 6 reports some use case steps, the SRL roles, and the generated constraints. For example, in S2, the CNP tool tags *"The system"* with *A0* (i.e., the actor performing the action), *"sets"* with verb, *"the occupant class for airbag control"* as *A1*, and *"to Init"* with *A2* (i.e., the final state). We ignore the prefix *"The system VALIDATES THAT"* in condition steps because it is not necessary to generate OCL constraints.

### 9.2 The OCL Generation Algorithm

We devised an algorithm that generates an OCL constraint from a given sentence in NL (see Fig. 7). We first execute the SRL toolset (the CNP tool) to annotate the sentence with the SRL roles (Line 2 in Fig. 7). We select and apply the transformation rules based on the verb in the sentence (Lines 3 to 7). The same rule is applied for all the verbs that are synonyms and that belong to the same VerbNet class. In addition, we have a special rule, hereafter we call *any-verb transformation rule*, that is shared by many verb classes. Each rule returns a candidate OCL constraint with a score assessing how plausible is the constraint (Section 9.2.5). We select the constraint with the highest score (Line 10).

For each verb, we classify the SRL roles into: (i) *entity role* indicating the entity whose state needs to be captured by the constraint, and (ii) *support roles* indicating additional information such as literals in the rhs-terms (see Fig. 6). We implemented our transformation rules according to the pairs $\langle$*entity role*, {*support roles*}$\rangle$ we extracted from the VerbNet role patterns. The role patterns provide all valid role combinations appearing with a verb.

Table 7 shows the role pairs for some of our transformation rules. For example, the verb 'to erase' has two VerbNet role patterns $\langle A0, V, A1\rangle$ and $\langle A0, V, A1, A2\rangle$ where $V$ is the verb, and *A0*, *A1* and *A2* are the SRL roles. The first

```
Require: srl, a sentence annotated with the different roles identified by SRL
Require: systemClass, the main class of the system
Require: rolesSetPairs, pairs of roles sets ⟨entity role, {support role}⟩
Ensure: ⟨ocl, score⟩, an OCL constraint with a score
 1 function TRANSFORM(srl, systemClass, rolesSetPairs)
 2  for each rolesSetPairs do
 3   lhsVariables ←process srl and identify a set of variables that might
 4                 appear in the left-hand side of the OCL constraint
 5   for each LHS in lhsVariables do
 6    RHS ← identify the term to put on the right-hand side
 7    OP ←identify the operator to use in the OCL constraint
 8    SEL ←if needed, build a subexpression with the selection operator
 9    QUERY ←identify the type of QUERY element
10    if RHS ≠ null and OP ≠ null then
11     ocl ←build the constraint using LHS, SEL, OP, RHS, and QUERY
12     score ←calculate the score of the OCL constraint
13     ocls ← ocls ∪ {⟨ocl, score⟩}
14    end if
15   end for
16  end for
17  bestOcl ← select the constraint with the best score from the list ocls
18  return bestOcl
19 end function
```

Fig. 8. The algorithm followed by each transformation rule.

```
Require: srl, a sentence annotated with the different roles identified by SRL
Require: systemClass, the main class of the system
Require: EntityRoles, list of entity roles
Require: SupportRoles, list of support roles
Ensure: Vars, list of left-hand side variables
 1 function FINDVARIABLES(srl, systemClass, EntityRoles, SupportRoles)
 2  for each ER in EntityRoles do
 3   term_ER ← preprocess(srl.get(ER))
 4   attrs ← findAttributes(systemClass, term_ER)
 5   Vars ← Vars ∪ attrs
 6   class ← findClass(term_ER)
 7   if class ≠ null then
 8    for each role in SupportRoles do
 9     attrs ← findAttributes(class, role)
10     Vars ← Vars ∪ attrs
11    end for
12   end if
13  end for
14  for each attr in Vars do
15   for each role in SupportRoles do
16    attrs ← extendByTraversingRelations(attr, role)
17    Vars ← Vars ∪ attrs
18   end for
19  end for
20  return Vars
21 end function
```

Fig. 9. The algorithm to identify lhs-variables.

pattern represents the case in which an object is erased (e.g., the measured voltage in S7 in Table 6), while, in the second one, an object is removed from a source (e.g., the occupant class being removed from the airbag control unit in S8). The transformation rule for the verb 'to erase' has thus two role pairs: ⟨*A1*, *null*⟩ and ⟨*A2*, {*A1*}⟩ (see Rule 4 in Table 7). Each transformation rule might be associated with multiple support roles; this is the case of the verb 'to set' whose role pair ⟨*A1*, {*A2*, *AM-LOC*}⟩ appears in Rule 3 in Table 7.

Each transformation rule performs the same sequence of activities for each pair ⟨*entity role*, {*support role*}⟩ (see Fig. 8). A rule first identifies the candidate lhs-variables (Line 3 in Fig. 8), and then builds a distinct OCL constraint for each lhs-variable identified (Lines 5 to 15). Finally, it returns the OCL constraint with the highest score (Line 18).

In Sections 9.2.1 to 9.2.5, we give the details of the algorithm in Fig. 8, i.e., identifying the lhs-variables (Line 3), selecting the rhs-terms (Line 6) and the OCL operators (Line 7), identifying the types of OCL query elements (Line 9), and scoring the constraints (Line 12). In this paper, we only provide the pseudocode of the function to identify lhs-variables, as it is the most complex one (Fig. 9). Interested readers can freely access the source code of the UMTG OCL generation component [111].

### 9.2.1 Identification of the Left-hand Side Variables

To identify the lhs-variables, the transformation rules follow an algorithm using the string similarity between the names of the domain model elements (i.e., the classes, attributes and associations) and the phrases in the use case step tagged with the entity and support roles (see Fig. 9). Based on Assumption 3, we expect that the phrase tagged with the entity role provides part of the information to identify the lhs-variable (e.g., *itsNVM* in S3 in Table 6), while the phrase(s) tagged with the support role further characterize the variable (e.g., *isAccessible* in S3).

The algorithm is influenced by domain modeling practices (Assumption 1). Assumptions A1.1 and A1.2 influence the criteria to select terms for the OCL constraint based on phrases in the use case step. Assumptions A1.1 - A1.5 influence the order in which noun phrases are processed.

TABLE 7
Entity and support roles for some transformation rules in UMTG.

| Rule ID | Verb | Entity roles | Support roles |
|---|---|---|---|
| 1 | to be | A1 | AM-PRD, AM-MNR, AM-LOC |
| 2 | to enable | A1 | AM-MNR |
| 3 | to set | A1 | A2, AM-LOC |
| 4 | to erase | A1 | |
| | | A2 | A1 |
| 5 | any verb | A1 | AM-PRD,Verb |

Based on A1.1, a domain model class best matches a phrase when its name shows the highest similarity with the phrase. To identify the matching classes and phrases, we employ the Needleman-Wunsch string alignment algorithm [112], which maximizes the matching between characters with some degree of discrepancy. For example, it enables matching composite nouns in the presence of plural words (e.g., the entity *TemperaturesDatabase* can be matched with the compound nouns *temperatures database* and *temperature databases*). However, we do not use the string alignment algorithm for attributes and associations because, in the context of embedded software development, attributes and associations often correspond to acronyms (e.g., RAM, ROM, and NVM) which are short and with a small alignment distance, but have different meanings. Based on A1.2, an attribute or association in the domain model best matches a given phrase (i) if it is a prefix or a postfix of the phrase, (ii) if it starts or ends with the phrase, or (iii) if it is a synonym or an antonym of the phrase. We ignore spaces and articles in the phrases. For each matching element, we compute a similarity score as the portion of matching characters.

The algorithm iterates over each phrase tagged with an entity role (Line 2 in Fig. 9). After the execution of *findAttributes* within each iteration, based on A1.2 and A1.4, we add to the list of the lhs-variables the system class attributes and associations that best match the phrase (Lines 4-5). In S3 in Table 6, *BodySense.itsNVM* is in the list of the lhs-variables because it terminates with *NVM* tagged with *A1*

(i.e., the entity role).

Based on A1.1 and A1.2, we look for the domain model class that best matches the phrase tagged with the entity role (Line 6). Based on A1.5, we recursively traverse the associations starting from this class to identify the related attributes that best match the phrases tagged with the support roles (Lines 8-11). The matching attribute might be indirectly related to the starting class. We give a higher priority to the directly related attributes. Therefore, the score of the matching attribute is divided by the number of traversed associations (Line 9). We add the best matching attributes to the list of the lhs-variables (Line 10). For example, for S2 in Table 6, the lhs-variable *BodySense.itsOccupancyStatus.occupantClassForAirbagControl* is identified by traversing the association *itsOccupancyStatus* from the system class *BodySense*. Its similarity score is 0.5 because one association has been traversed (*itsOccupancyStatus*) and there is an exact match between the attribute name and the noun phrase tagged with *A1*.

We further refine the lhs-variables with a complex type (i.e., a class or a data type). For each attribute and association in the list of the lhs-variables, we traverse the related associations to identify the attributes and associations that best match the phrases tagged with the support roles (Lines 14 to 19). In S3, *BodySense.itsNVM* is refined to *BodySense.itsNVM.isAccessible* since the class *NVM* has a boolean attribute (*isAccessible*) with a name similar to the phrase tagged with *AM-PRD* (*accessible*).

Based on A1.3, when the phrase tagged with the entity role includes a possessive or attributive phrase, we look for attributes/associations in the domain model that reflect the relation between the possessive/attributive phrase and the main phrase in the entity role phrase (e.g., *the watchdog* and *counter* in sentence S12 in Table 6) . We rely on the NomBank tags generated by CNP to identify the main phrase and the possessive/attributive phrases. In the domain model, the main phrase usually best matches an attribute/association that belongs to an *owning entity*. The owning entity is either (1) a class that best matches the possessive/attributive phrase or (2) a class referred by an attribute of the main system class that best matches the possessive/attributive phrase. For example, in the case of S13, the attribute *counter* (i.e., the main phrase in S13) belongs to the entity class referenced by the attribute *watchdog* (i.e., the possessive/attributive phrase) of the main system class. Based on this observation, to identify the model elements that reflect the relation captured by a possessive/attributive phrase, we perform an additional execution of the function *findVariables* where we treat possessive/attributive phrases as the entity role and the main phrase as a support role. The possessive/attributive phrase is thus used to identify the owning entity while the main phrase is used to identify the owned attribute/association. In the case of S13, UMTG looks for an attribute of the system class that best matches *watchdog* and then further refines the search by looking for a contained attribute that best matches *counter*.

#### 9.2.2 Identification of the Right-hand Side Terms

The rhs-term can be a literal or a variable. It is identified based on the lhs-variable and on the support roles that have not been used to select the lhs-variables. If the lhs-variable is of a boolean or numeric type, the rhs-term is a boolean or numerical value derived from a phrase tagged with one of the support roles. Therefore, we look for a phrase that matches the terms 'true' and 'false' or that is a number.

When the lhs-variable is boolean and the verb is negative, we negate the rhs-term. When the lhs-variable is boolean and there is no support role to identify the rhs-term, we use the default value *true* for the rhs-term. For example, in S4 in Table 6, all the support roles have already been used to identify the lhs-variable and there is no support role left for the rhs-term. Therefore, we use *true* for the rhs-term.

When the lhs-variable is of an enumeration type, we identify the enumeration literal in the domain model that best matches the phrases tagged with the support roles. For instance, this is the case for S5 in Table 6, which results from the application of the *any-verb* transformation rule. In S5, *BodySense.buildCheckStatus* is the lhs-variable which is of an enumeration type, i.e., *BuildCheckStatus*. This enumeration contains a term that matches the root of the verb in S5 (*pass*), and the verb is one of the support roles in the *any-verb* transformation rule. Therefore, the literal *BuildCheckStatus::Passed* is selected as the rhs-term.

For certain transformation rules, the rhs-term is set to a specific literal based on the meaning of the verb. This is the case for the verb *erase*, used in sentences S7 and S8 in Table 6. For *erase*, the literal to be used is selected based on the type of the lhs-variable (i.e., `0` for numeric types and `null` for complex types).

#### 9.2.3 Identification of the OCL operators

The OCL comparison operators are identified based on the verb in the use case step. The operator '=' is used for most verbs. For the verb '*to be*', we rely on Roy et al. [113] which, for example, identify the operator '>' for *"capacitance is above 600"*. More precisely, we apply Roy's approach on the phrase tagged with the support-role used to identify the rhs-term. If antonyms have been used to identify the lhs-variable and the rhs-term, we negate the selected operator, for instance, by replacing '=' with '! =' in S9 in Table 6.

Regarding the selection operator, we are limited to the pattern in Fig. 6, i.e., the exclusion of some class instances in a set. The selection operator is introduced when the phrase tagged with *A1* contains the keyword *except* (e.g., S6 in Table 6) or any of its synonyms according to WordNet (i.e., *leaving out*, *excluding*, *leaving off*, *omitting*, and *taking out*). To identify the types of the instances to be excluded, we rely on the tags generated based on SRL NomBank. We look for the phrases tagged with *A2* to identify the adverbial clause. We identify all the distinct noun phrases within the clause (e.g., *voltage errors* and *memory errors* in S6).

#### 9.2.4 Identification of the Type of OCL Query Elements

The query elements in our OCL pattern are used to check if an expression holds for a set of instances (see Fig. 6). The key difference among them is the number of instances for which the expression should hold. Since the number of subjects referred to in a sentence in English is specified by the determiners and quantifiers, we identify the type of the query element based on the determiners and quantifiers in the phrase tagged with an entity role. We consider entity roles

because a domain entity is selected based on its similarity with the phrase tagged with an entity role.

In English, the indefinite articles *a* and *an* and the determiner *some* refer to particular members of a group. Therefore, if a phrase tagged with an entity role includes an indefinite article or the determiner *some*, we generate an OCL query following the EXISTS sub-pattern (see Fig. 6). For phrases with a quantifier referring to a certain number of members of a group (e.g., *at least five*), we generate expressions following the COUNT sub-pattern. We also rely on Roy's approach [113] to generate a quantifier formula with an operator and a numeric literal.

For all other cases, we generate expressions that follow the FORALL sub-pattern. These cases include phrases with universal determiners referring to all the members of a group (e.g., *any*, *each*, and *every*) and phrases with the definite article *the*. The definite article is used to refer to a specific entity (e.g., *the measured voltage* in S7 in Table 6) and typically leads to the definition of an attribute or association in the domain model (e.g., *the measured voltage* matches an attribute of the system class `BodySense`). Since there might be multiple class instances with the corresponding attribute (or association), we rely on the FORALL sub-pattern to match all of them. When the definite article is used to refer to a specific instance among all the instances of the same type, engineers should design the domain model to enable the identification of such instances. For example, to identify the last measurement in the capacitance sequence recorded by *BodySense*, which is referred to in S14, engineers introduced the attribute *last measurement*, which is a derived attribute. A detailed description of how the universal determiners and the definite article influence the scoring of the generated constraints is provided in Section 9.2.5.

Our OCL constraint generation strategy does not support the generation of OCL constraints for fault tolerant devices including replicated subsystem instances (e.g., multiple instances of BodySense). More precisely, UMTG cannot derive OCL constraints for subsystems with different states (e.g., one instance of BodySense with a temperature error being detected and another one with no error detected). This limitation is inherited from RUCM, which does not include means to capture the behavior of multiple subsystems because, in the requirements elicitation phase, analysts describe the system as a whole independently from the number of implemented instances. Accordingly, we use OCL constraints to capture the properties of the system as a whole, not the properties of its subsystems.

#### 9.2.5 Scoring of the OCL Constraints

The score of an OCL constraint accounts for both the completeness and correctness of the constraint. Completeness relates to the extent to which all the concepts in the use case step are accounted for. Correctness relates to how similar the variable names in the constraint are to the concepts in the use case step.

We measure the completeness of a constraint in terms of the percentage of the roles in the use case step that are used to identify the terms in the constraint.

To compute the correctness of a constraint, we use $correctness = \frac{(lhsScore + rhsScore + matchUniversalDeterminer)}{3}$ where *lhsScore* and *rhsScore* are the similarity scores for the lhs-variable and the rhs-term, respectively. *lhsScore* is the average of the scores of all the attributes/associations in the variable. When the rhs-term is a boolean, numeric or enumeration literal generated from a phrase in the use case step, *rhsScore* is set to one; otherwise, *rhsScore* is computed like *lhsScore*. *matchUniversalDeterminer* is 1 when universal determiners (e.g., *any*, *every*, or *no*) are properly reflected in the constraint. We consider a universal determiner to be properly reflected in the constraint (1) when it is not in the noun phrase tagged with an entity role and the constraint refers to a specific instance associated with the system class (e.g., S3), and (2) when it is in the noun phrase tagged with an entity role and the constraint refers to all the instances of the class matching the phrase (e.g., S6). Universal determiners are important to derive the correct constraints. For example, for S3, we build two constraints *C2* and *C3* in Table 8. We select *C3* because the use case step does not explicitly indicate that all the NVM components should be considered.

The final score is computed as the average of the completeness and correctness scores (see Table 8).

TABLE 8
OCL constraints and their scores for sentence S3 of Table 6.

| | Candidate OCL | Score |
|---|---|---|
| C1 | $BodySense.allInstances() \rightarrow forAll(i \mid i.itsNVM = ...)$ | – |
| C2 | $NVM.allInstances() \rightarrow forAll(i \mid i.isAccessible = true)$ | 0.81 |
| C3 | $BodySense.allInstances() \rightarrow forAll(i \mid i.itsNVM.isAccessible = true)$ | 0.94 |

**Notes on the computed scores:**
**C1** is ignored because the attribute *BodySense.itsNVM* refers to a class and does not enable the identification of any rhs-term. In **C2**, *completeness* is 1 since all the roles are used to identify the terms. *lhsScore* is 0.83, i.e., the division of the length of the word *accessible* (i.e., 10) by the length of the variable *isAccessible* (i.e., 12). *rhsScore* is 1. *matchUniversalDeterminer* is 0 because the constraint refers to all the instances of class NVM although no universal determiner is used in the sentence. *correctness* is calculated as $(0.83 + 1 + 0)/3 = 0.61$. In **C3**, *completeness* is 1. *lhsScore* is 0.66, which is the average of the scores for attributes *itsNVM* (i.e., 0.5) and *isAccessible* (i.e., 0.83). *rhsScore* is 1. *matchUniversalDeterminer* is 1 because no universal determiner is used in the sentence and the constraint refers to a specific instance referenced by the system class. *correctness* is calculated as $(0.66 + 1 + 1)/3 = 0.89$. The score is thus calculated as $(0.89 + 1)/2 = 0.94$.

TABLE 9
Verbs unlikely to appear in use case specifications.

| Categories of Verbs for Exclusion | Example Verbs |
|---|---|
| Verbs describing a human feeling | love, like |
| Verbs describing a human sense | smell, taste |
| Verbs describing human behaviors | wish, hope, wink, cheat, confess |
| Verbs describing body internal motion | giggle, kick |
| Verbs describing body internal states | quake, tremble |
| Verbs describing manner of speaking | burble, croak, moan |
| Verbs describing nonverbal expressions | scoff, whistle |
| Verbs describing animal behaviors | bark, woof, quack |
| Communication verbs | tell, talk |

### 9.3 Correctness and Generalizability

The adoption of verb-specific transformation rules may limit the correctness and generalizability of constraint generation due to the considerable number of English verbs. For example, the *Unified Verb Index* [114], a popular lexicon based on VerbNet and other lexicons, includes 8,537 English verbs.

To strengthen the correctness of constraint generation, we base our transformation rules on the VerbNet role pat-

terns. These role patterns capture all valid role combinations appearing with a verb. The rules are applied according to the entity and support role pairs $\langle$*entity role*, $\{$*support roles*$\}\rangle$ we extracted from the role patterns (see Table 7). For example, the transformation rule of the verb 'to erase' has two role pairs $\langle A1, null\rangle$ and $\langle A2, \{A1\}\rangle$ extracted from the VerbNet role patterns (see Rule 4 in Table 7). If the sentence contains *A1* without *A2*, then *A1* is used to identify the entity to be erased, i.e., the attribute in the lhs-variable (see S7 in Table 6). If the sentence contains both *A1* and *A2*, then *A2* is used to identify the entity and *A1* provides some additional information (i.e., the attribute of the entity to be erased). As an example of the latter case, in S8 in Table 6, we identify the entity name (*AirbagControlUnit*) and the attribute in the lhs-variable (*occupantClass*). In addition to the case studies in our evaluation (see Section 14), we validated our transformation rules with sentences that include all the role patterns in VerbNet.

To ensure the generalizability of the constraint generation, we employ three key solutions which prevent the implementation of hundreds of rules. First, we rely on the VerbNet classes to use a single rule targeting different verbs. Since all the verbs in a VerbNet class share the same role patterns, we reuse the same rule for all the verbs in the VerbNet class that are synonyms according to WordNet.

Second, we excluded some VerbNet classes of verbs, i.e., 225 classes and 175 subclasses, that are unlikely to appear in specifications (see Table 9). We manually inspected all the VerbNet classes to identify them (e.g., verbs describing human feelings). Our analysis results are available online [115].

Third, we further analyzed the remaining classes to determine the verbs that are processed by our *any-verb transformation rule*. This analysis shows only 33 verb-specific rules are required to process 87 classes of verbs.

## 10 Generation of Use Case Test Models

UMTG generates a *UCTM* from an RUCM use case specification along with the generated OCL constraints (Step 7 in Fig. 1). The model makes the implicit control flow in the use case specification explicit and maps the use case steps onto the test case steps. Fig. 10 gives the metamodel for use case test models.

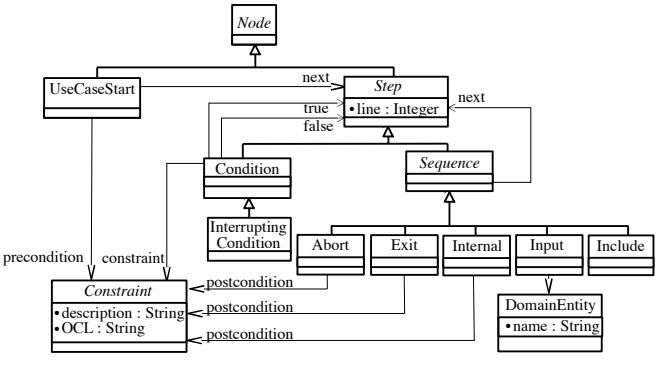


Fig. 10. Metamodel for use case test models.

A use case test model is a connected graph containing the instances of multiple subclasses of *Node*. *UseCaseStart* represents the beginning of a use case with a precondition and is linked to the first step in the use case (*next*). There are two subclasses of *Step*, i.e., *Sequence* and *Condition*. *Sequence* has a single successor. *Condition* is linked to a constraint (*constraint*) and has two possible successors (*true* and *false*). *InterruptingCondition* enables the execution of a global or bounded alternative flow. *Constraint* has a condition and an OCL constraint generated from the condition.

*Input* indicates the invocation of an input operation and is linked to *DomainEntity* that represents input data. To specify the effects of an internal step on the system state, *Internal* is linked to *Constraint* (i.e., *postcondition*). *Exit* represents the last step of a use case flow; *Abort* indicates the termination of an anomalous flow. *Exit* and *Abort* steps have an associated *Constraint* with a postcondition in textual form, which is used by UMTG to generate oracles (see Section 12). In the case of *Exit* and *Abort* steps, UMTG does not generate an OCL constraint from the textual condition (see Section 9).

Output steps are not represented in use case test models because UMTG does not rely on them to drive test generation. Although output steps might be used to select outputs to be verified by automated oracles during testing, UMTG relies on postconditions to generate test oracles because they provide more information (e.g., the system state and the value of an output, which are not indicated in output steps).

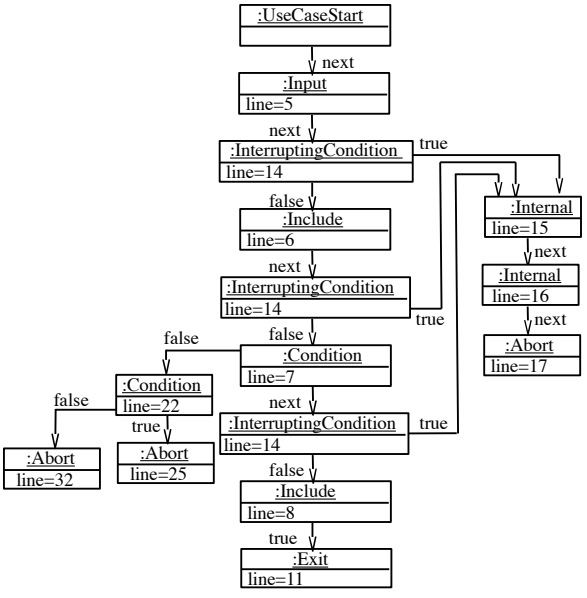


Fig. 11. Part of the use case test model for *Identify Occupancy Status*.

Fig. 11 shows part of the use case test model generated from the use case *Identify Occupancy Status* in Table 4. UMTG processes the use case steps annotated by the NLP pipeline in Section 7. For each textual element tagged with *Input*, *Include*, *Internal*, or *Condition*, a *Step* instance is generated and linked to the previous *Step* instance. For each domain entity in the use case specification, a *DomainEntity* instance is created and linked to the corresponding *Step* instance.

For each specific alternative flow, a *Condition* instance is generated and linked to the *Step* instance that represents the first step of the alternative flow (e.g., the *Condition* instance for Line 7 in Fig. 11).

Global and bounded alternative flows begin with a guard condition and are used to indicate that the condition might become true at run time and trigger the execution of the alternative flow (e.g., as an effect of an interrupt being triggered). For each step referenced by a global or bounded alternative flow, an *InterruptingCondition* instance is created and linked to the *Step* instance that represents the reference flow step (e.g., the three *InterruptingCondition* instances for Line 14 linked to the *Step* instances for Lines 6, 7, and 8 in Fig. 11).

For multiple alternative flows depending on the same condition, *Condition* instances are linked to each other in the order the alternative flows appear in the specification. For an alternative flow in which the execution is resumed back to the basic flow, an *Exit* instance is linked to the *Step* instance that represents the reference flow step.

## 11 Generation of Use Case Scenarios and Test Inputs

UMTG generates, from a use case test model along with OCL constraints, a set of use case scenarios and test inputs (Step 8 in Fig. 1). A scenario is a path in the use case test model that begins with a *UseCaseStart* node and ends with an *Abort* node or an *Exit* node with the attribute *next* set to null (i.e., an Exit node terminating the use case under test). It captures a sequence of interactions that should be exercised during testing. It consists of a finite number of steps. However, it can exercise alternative flows that lead to loops in use case specifications. More precisely, a scenario can exercise the same use case step up to $T$ times in a loop, with $T$ being a UMTG parameter specified by engineers.

UMTG needs to identify test inputs in which the conditions in the scenario hold. For example, to test the scenario for the basic flow of the use case *Classify Occupancy Status* in Table 4, we need test inputs in which the capacitance value is above 600 (see Line 67 in Table 4). We use the term *path condition* to indicate a formula that conjoins all the conditions (OCL constraints) in a given scenario. If the path condition is satisfiable, we derive an object diagram (i.e., an instance of the domain model in which the path condition holds). For a given scenario, test input values are extracted from the object diagram that satisfies the path condition.

We devise an algorithm, *GenerateScenariosAndInputs* in Fig. 12, which generates a set of pairs of use case scenarios and object diagrams from the input use case test model $tm$. Before calling *GenerateScenariosAndInputs*, the use case test model $tm$ is merged with the use case test models of the included use cases in a way similar to the generation of interprocedural control flow graphs [116]. The *Include* instances in $tm$ are replaced with the *UseCaseStart* instances of the included use cases; the *Exit* instances of the basic flows of the included use cases are linked to the *Node* instances following the *Include* instances in $tm$.

To generate use case scenarios, we call the function *GenerateScenarios* with *tm* being a use case test model, *Scenario* being an empty list, *tm.start* being the *UseCaseStart* instance in $tm$, *pc* being a *null* path condition, *ScenariosInputs* being an empty list, and *ScenariosSet* being an empty list (Line 7 in Fig. 12). UMTG employs the Alloy analyzer [38] to generate an object diagram in which the path condition in OCL holds

**Require:** $tm$, a use case test model
**Ensure:** $ScenariosInputs$, a set of pairs $\langle Scenario, objectDiagram\rangle$
1 **function** GenerateScenariosAndInputs(tm)
2 $Scenario \leftarrow newList()$ //list of Nodes
3 $pc \leftarrow newList()$ //list of Constraints
4 $ScenariosInputs \leftarrow newSet()$ //set with feasible scenarios
5 **repeat**
6 $ScenariosSet \leftarrow newSet()$ //set of pairs $\langle Scenario, pc\rangle$
7 $GenerateScenarios(tm, tm.start, Scenario, pc, ScenariosInputs, ScenariosSet)$
8 **for each** pair $\langle Scenario, pc\rangle$ *in* $ScenariosSet$ **do**
9 $objectDiagram \leftarrow Solve(Scenario, pc)$
10 **if** $objectDiagram = null$ **then** //the scenario is infeasible
11 *remove* $\langle Scenario, pc\rangle$ *from* $ScenariosSet$
12 **else**//add the scenario to the results to be returned
13 $ScenariosInputs \leftarrow$
14 $ScenariosInputs \cup \{\langle Scenario, objectDiagram\rangle\}$
15 **end if**
16 **end for**
17 **until** $coverageSatisfied(tm, ScenariosInputs)$ **or** *max number of iterations T reached*
18 **if** coverage criterion is subtype coverage **then**
19 $ScenariosInputs \leftarrow maximizeSubTypeCoverage(ScenariosInputs)$
20 **end if**
21 **return** $ScenariosInputs$
22 **end function**

Fig. 12. Algorithm for generating use case scenarios and test inputs.

for a given scenario (Line 9). It makes use of existing model transformation technology from OCL constraints to Alloy specifications [117].

Some of the generated scenarios may be infeasible. These are the scenarios that cannot be exercised with any set of possible values such as scenarios covering auxiliary basic and alternative flows. For example, in Table 4, this is the case for the scenario that covers both the alternative flow that detects a low temperature error (Line 51) and the basic flow of Classify Occupancy Status (Line 64). Based on the condition step in Line 7, the basic flow of Classify Occupancy Status can be executed only if no error has been detected. We exclude such infeasible scenarios (Lines 10 and 11).

We execute *GenerateScenarios* multiple times until a selected coverage criterion is satisfied or the number of iterations reaches a predefined threshold (Line 17). We set the threshold to ten in our experiments. UMTG supports three distinct coverage criteria, i.e., branch coverage, a lightweight form of def-use coverage, and a form of clause coverage that ensures each condition to be covered with different entity types. All these three coverage strategies are described in the following sections.

Section 11.1 describes the scenario generation algorithm (Line 7), while Section 11.2 provides details about the generation of object diagrams that satisfy path conditions (Line 9).

### 11.1 Generation of Use Case Scenarios

*GenerateScenarios* performs a recursive, depth-first traversal of a use case test model to generate use case scenarios (see Fig. 13). It takes as input a use case test model (*tm*), a node in *tm* to be traversed (*node*), two input lists (i.e., *Scenario* and *pc*) and two input sets (i.e., *Curr* and *Prev*). *Scenario* is a list of traversed nodes in *tm*, and *pc* is the path condition of the traversed nodes. *Prev* is a set of feasible scenarios traversed in previous invocations of *GenerateScenarios* made by *GenerateScenariosAndInputs*. *Prev* corresponds to *ScenariosInputs* in *GenerateScenariosAndInputs*. *Curr* is a set of pairs

**Require:** $tm$, a use case test model
**Require:** $node$, an instance in $tm$
**Require:** $Scenario$, a linked list of node instances from $tm$
**Require:** $pc$, an OCL constraint of the path condition for $Scenario$
**Require:** $Prev$, a list of scenarios covered in previous iterations
**Ensure:** $Curr$, a set of pairs $\langle Scenario, pc \rangle$

```
1  function GENERATESCENARIOS(tm, node, Scenario, pc, Prev, Curr)
2   if (coverageSatisfied(tm, Prev, Curr)) then
3    return
4   end if
5   if (unsatisfiable(pc)) then
6    return
7   end if
8   if (node is a UseCaseStart instance) then
9    addToScenario(node, Scenario)
10   pc ← composeConstraints(pc, node.precondition.ocl)
11   GenerateScenarios(tm, node.next, Scenario, pc, Prev, Curr)
12  end if
13  if (node is an Input instance) then
14   addToScenario(node, Scenario)
15   GenerateScenarios(tm, node.next, Scenario, pc, Prev, Curr)
16  end if
17  if (node is a Condition but not an InterruptingCondition then)
18   //prepare for visiting the true branch
19   Scenario_T ← Scenario //create a scenario copy
20   addToScenario(node, Scenario_T)
21   pc_T ← composeConstraints(pc, node.constraint.ocl)
22   GenerateScenarios(tm, node.true, Scenario_T, pc_T, Prev, Curr)
23   //prepare for visiting the false branch
24   Scenario_F ← Scenario //create a scenario copy
25   addToScenario(node, Scenario_F)
26   nc ← negateConstraint(node.constraint.ocl)
27   pc_F ← composeConstraints(pc, nc)
28   GenerateScenarios(tm, node.false, Scenario_F, pc_F, Prev, Curr)
29  end if
30  if (node is an InterruptingCondition instance) then
31   //visit the false branch first, i.e., a scenario without any interrupt
32   Scenario_F ← Scenario //create a scenario copy
33   GenerateScenarios(tm, node.false, Scenario_F, pc, Prev, Curr)
34   //visit the true branch, i.e., a scenario with an interrupt
35   Scenario_T ← Scenario //create a scenario copy
36   addToScenario(node, Scenario_T)
37   pc_T ← composeConstraints(pc, node.constraint.ocl)
38   GenerateScenarios(tm, node.true, Scenario_T, pc_T, Prev, Curr)
39  end if
40  if (node is an Internal instance) then
41   addToScenario(node, Scenario)
42   pc ← composeConstraints(pc, node.postcondition.ocl)
43   GenerateScenarios(tm, node.next, Scenario, pc, Prev, Curr)
44  end if
45  if (node is an Exit or Abort instance) then
46   if (node is an Exit instance with a non null next association) then
47    if (node.next visited at most T times in the current Scenario) then
48     //the visit should proceed in the specified next step
49     addToScenario(node, Scenario)
50     GenerateScenarios(tm, node.next, Scenario, pc, Prev, Curr)
51    else//node.next visited more than T times in the current scenario
52     return //ignore paths that traverse a same branch or loop body more
             than T times
53    end if
54   else //is the final step of the use case
55    addToScenario(node, Scenario)
56    if (coverageImproved(tm, Curr, Scenario)) then
57     Curr ← Curr ∪ {⟨Scenario, pc⟩}
58    end if
59   end if
60  end if
61  return
62 end function
```

Fig. 13. Algorithm for generating use case scenarios.

$\langle Scenario, pc \rangle$ which contain the scenarios and their path conditions identified during the current invocation of *GenerateScenarios* made by *GenerateScenariosAndInputs*. *Scenario*, *pc*, and *Curr* are initially empty. They are populated during recursive calls to *GenerateScenarios*.

Fig. 14 shows three scenarios generated from the use case test model in Fig. 11. Scenario A covers, without taking any alternative flow, the basic flow of the use case *Identify Occupancy Status* and the basic flows of the included use cases *Self Diagnosis* and *Classify Occupancy Status* in Table 4. Scenario B takes two specific alternative flows of the use cases *Self Diagnosis* and *Identify Occupancy Status*, respectively. It covers the case in which a TemperatureError has been detected (Lines 51 to 55 in Table 4) and some error has been qualified (Lines 20 to 27). Scenario C covers the case in which some error has been detected but no error has been qualified (Lines 28 to 33).

The precondition of the *UseCaseStart* instance is added to the path condition for the initialisation of the test case (Lines 8 to 12 in Fig. 13). *Input* instances do not have associated constraints to be added to the path condition (Lines 13 to 16). We recursively call *GenerateScenarios* for the nodes following the *UseCaseStart* and *Input* instances (Lines 11 and 15). For instance, Scenario A in Fig. 14 starts with the *UseCaseStart* instance of the use case *Identify Occupancy Status* followed by an *Input* instance, and proceeds with the *UseCaseStart* instance of the included use case *Self Diagnosis*.

For each *Condition* instance which is not of the type *InterruptingCondition*, we first visit the true branch and then the false branch to give priority to nominal scenarios (Lines 17 to 29). In the presence of a coverage-based stopping criterion, prioritizing nominal scenarios is useful to generate relatively short use case scenarios covering few alternative flows (i.e., what happens in realistic executions) instead of long use case scenarios covering multiple alternative flows. Bounded and global alternative flows are taken in the true branch of their *InterruptingCondition* instances. Therefore, to give priority to nominal scenarios, we first visit the false branch for the *InterruptingCondition* instances (Lines 30 to 39). For each condition branch taken, we add to the path condition the OCL constraint of the branch (Lines 21, 27 and 37) except for the false branch of the *InterruptingCondition* instances; indeed, the condition of a bounded or global alternative flow is taken into account only for the scenarios in which the alternative flow is taken. For example, Scenarios A, B, and C do not cover the condition of the bounded alternative flow of the use case *Identify Occupancy Status*. We call *GenerateScenarios* for each branch (Lines 22, 28, 33 and 38).

The *Internal* instances represent the use case steps in which the system alters its state. Since state changes may affect the truth value of the following *Condition* instances, the postconditions of the visited *Internal* instances are added to the path condition (Line 42). We call *GenerateScenarios* for the nodes following the *Internal* instances (Line 43).

To avoid infinite cycles in *GenerateScenarios*, we proceed with the node following an *Exit* instance only if the node has been visited at most a number of times $T$ specified by the software engineer (Lines 46 and 47). The default value for $T$ is one, which enables UMTG to traverse loops once. An *Exit* instance followed by a node represents either a resume step or a final step of an included use case. An *Exit* or *Abort* instance which is not followed by any node indicates the termination of a use case. For instance, Scenario A terminates with the *Exit* instance of the basic flow of the use case *Identify Occupancy Status*, while Scenario B terminates with the *Abort* instance of the first specific alternative flow of the same use case. When the current scenario ends, we

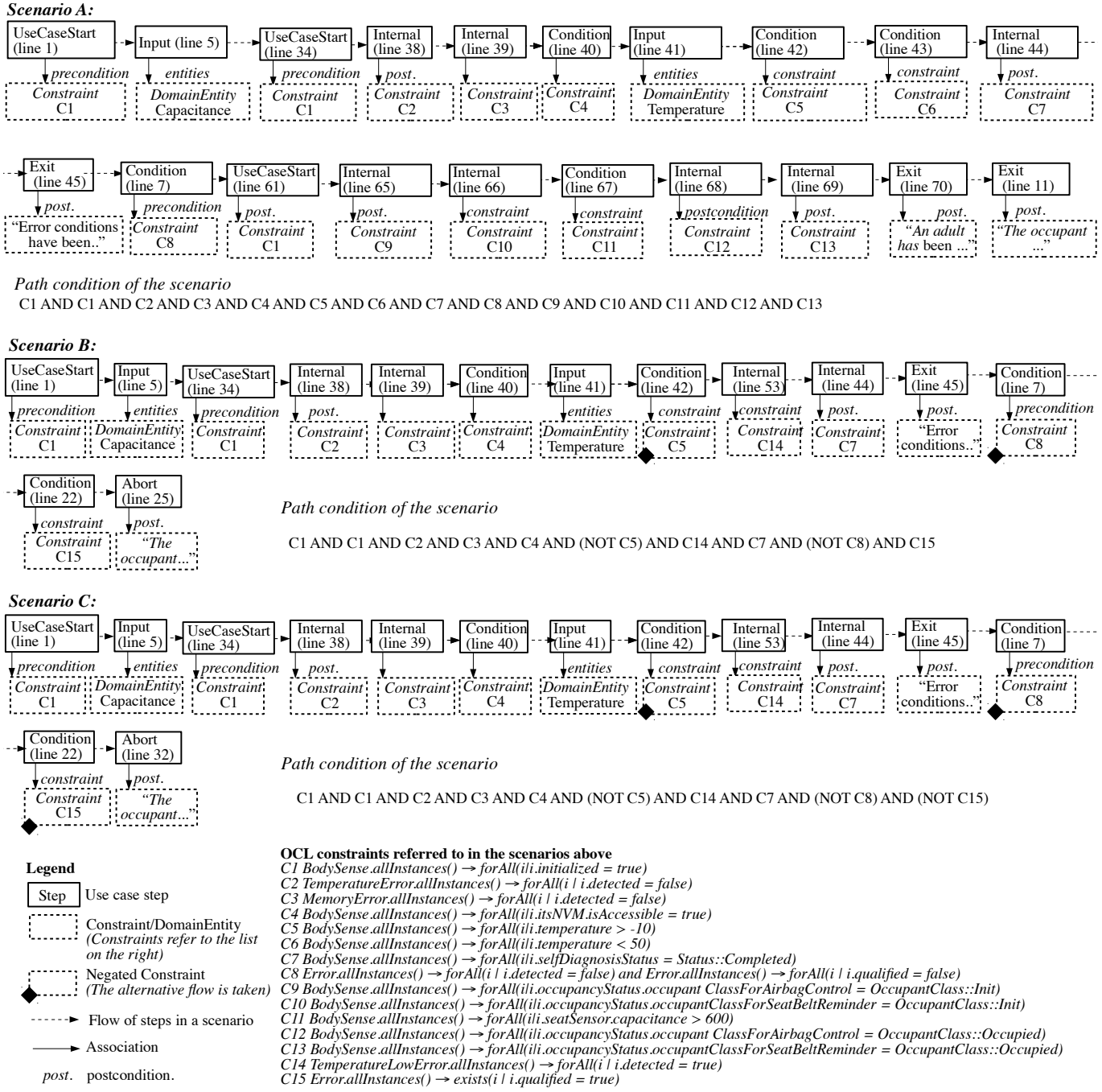


Fig. 14. Three scenarios built by UMTG when generating test cases for the use case 'Identify Occupancy Status' in Table 4.

add the scenario and its path condition to the set of pairs of scenarios and path conditions to be returned (Line 57), if it improves the coverage of the model based on the selected coverage criterion (Line 56).

The algorithm is guided towards the generation of scenarios with a satisfiable path condition (Lines 5 to 7). To limit the number of invocations of the constraint solver and, consequently, to speed up test generation, the function *unsatisfiable* (Line 5) does not execute the constraint solver to determine satisfiability but relies on previously cached results. It returns *false* if the path condition has been determined to be unsatisfiable by the solver during the generation of test inputs (Line 9 in Fig. 12); otherwise it assumes that the path condition is satisfiable and returns *true*. This is why *GenerateScenarios* is invoked multiple times by *GenerateScenariosAndInputs* and constraint solving is executed after all the path conditions have been collected (Line 9 in Fig. 12). Multiple invocations of *GenerateScenarios* might traverse already visited scenarios, except for the ones including an unsatisfiable path condition. Based on our experience, this does not lead to any major scalability drawback. Optimizations, such as avoiding the traversal of fully visited portions of the UCTM, are part of our future work.

Scenario generation terminates when the selected coverage criterion is satisfied (Lines 2 to 4) or the entire graph

is traversed. The depth-first traversal ensures that all edges can be visited at least once. UMTG supports three coverage criteria: branch coverage, a lightweight form of def-use coverage, and a form of clause coverage that we call subtype coverage.

The *branch coverage criterion* aims to maximize the number of branches (i.e., edges departing from condition steps) that have been covered. Coverage improves (Line 56 in Fig. 13) any time a scenario covers a branch that was not covered yet.

Our *def-use coverage criterion* aims to maximize the coverage of entities referred (i.e., used) in condition steps that present attributes defined in internal steps. We identify *definitions* from the postconditions associated to internal steps. More precisely, each postcondition generally defines one entity, i.e., one that owns the lhs-variable appearing in the postcondition; postconditions that join multiple OCL constraints using the operators *and*/*or* can define multiple entities. Sentence S4 in Table 6 defines one entity, *TemperatureError*. We identify *uses* from condition steps' constraints; used entities are the ones that own the terms appearing in condition steps' constraints. More formally, our coverage criterion matches the standard definition of *all p-uses* [118] except that we treat positive and negative evaluation of predicates as different def-use pairs. This is done to enforce the pairing of each definition with both the true and false evaluation of the constraints in which it is used. The def-use pairs covered by each scenario are computed by traversing the scenario and by identifying all the pairs of use case steps $\langle s_d, s_u \rangle$, where $s_d$ defines an entity and $s_u$ uses the same entity or one of its supertypes. Def-use coverage improves any time a new def-use pair is observed in a scenario.

Def-use coverage leads to scenarios not identified with branch coverage. In our running example, by maximizing the branch coverage criterion we ensure that both Scenario B and Scenario C in Fig.14 are covered. Indeed, the entity *TemperatureLowError* is defined in Line 53 and used by the constraint C15, which is referenced in the condition step in Line 22; this leads to two def-use pairs, one covered by Scenario B (C15 evaluates to true) and one by Scenario C (C15 evaluates to false). In addition to these two scenarios, the def-use coverage criterion also ensures that a *TemperatureHighError* is covered in two additional scenarios (not shown in Fig.14) that pair the definition of *TemperatureHighError* in Line 59 with its uses in Line 22.

The *subtype coverage criterion* aims to maximize the number of entity (sub) types with an instance referenced in a satisfied condition. For each scenario, it leads to a number of object diagrams such that each condition is satisfied with all the possible entity (sub) types referenced by the condition. For example, in the presence of a condition step referring to the generic entity type *Error*, it generates a number of object diagrams such that, for each subtype of *Error*, there is at least one instance satisfying the condition.

To apply the *subtype coverage criterion*, the algorithm *GenerateScenariosAndInputs* further processes the generated scenarios with function *maximizeSubTypeCoverage* (Line 19 in Fig. 12). Function *maximizeSubTypeCoverage* appears in Fig. 15. It identifies all the constraints appearing in condition steps that are evaluated to true (Line 5 in Fig. 15). For each of these constraints, it re-executes constraint solving multiple

**Require:** $ScenariosInputs$, a set of pairs $\langle Scenario, objectDiagram \rangle$
**Ensure:** $AugmentedInputs$, set of pairs $\langle Scenario, objectDiagram \rangle$ that maximize the subtype coverage metric

```
1  function MAXIMIZESUBTYPECOVERAGE(ScenariosInputs)
2    AugmentedInputs ← copy(SecenariosInputs)
3    for each ⟨Scenario, objectDiagram⟩ in ScenariosInputs do
4     for each ⟨step⟩ in Scenario do
5      if (node is a Condition instance evaluated to true) then
6       subtypes ← subtypesOf(identifyUsedEntity(step))
7       for each sT in subtypes do
8        newObjDiagr ← solveWithSubtype(Scenario, sT)
9        if (coverageImproved(newObjDiagr, AugmentedInputs)) then
10         AugmentedInputs ←
11                  AugmentedInputs ∪ ⟨Scenario, newObjDiagr⟩
12       end if
13      end for
14     end if
15    end for
16   end for
17   return AugmentedInputs
18 end function
```

Fig. 15. Function to maximize subtype coverage.

times, once for each type *sT* that is a subtype of the entity used by the constraint (Line 6). Constraint solving is forced to generate a solution that contains at least one instance of the subtype *sT* that satisfies the constraint (Line 8). For each scenario, we keep only the object diagrams that contain assignments not observed before (implemented by function *coverageImproved*, see Lines 9-11).

In our example, to maximize subtype coverage, we need to cover *C15* in Scenario B in Fig. 14 with three instances of subtypes of the *Error* entity: *TemperatureLowError* (i.e., an instance of *TemperatureLowError* should have the attribute *qualified* set to true), *TemperatureHighError*, and *VoltageError*. In Scenario B, to ensure that *C15* is covered with an instance of *TemperatureLowError*, the function *solveWithSubType* (Line 8 in Fig. 15) extends the path condition of the scenario with the additional condition $TemperatureLowError.allInstances() \rightarrow exists(i|i.qualified = true)$, which is automatically derived from constraint *C15* by replacing the entity type used in the constraint. The same is repeated also for *TemperatureHighError* and *VoltageError*. In our implementation, the subtype coverage criterion is combined with the def-use coverage criterion to ensure that constraints using input types are exercised with all the possible input subtypes. In our running example, the subtype coverage metric ensures that Scenario B is covered with a *TemperatureLowError* being both detected and qualified.

The detailed *time complexity analysis* for *GenerateScenariosAndInputs* is provided in Appendix A. In the worst case, the generation of use case scenarios is at least quintic with respect to the number of nodes in the use case test model (UCTM), i.e., $\mathcal{O}(N^{T+4})$. In the average case, we expect the complexity to be cubic, i.e., $\mathcal{O}(N^3)$. However, we observe that, based on our empirical evaluation, the generation of use case scenarios with UMTG is feasible in practical time. The main explanation is that the number of nodes in UCTMs is usually not high since use case specifications, in practice, tend to describe high-level system behavior, e.g., system-actor interactions. For example, the largest UCTM in our case studies, which has been generated for *BodySense*, includes 154 nodes. With a relatively small number of nodes, UCTMs can be processed fast by using modern hardware

(e.g., laptops), even in the presence of $\mathcal{O}(N^3)$ complexity.

### 11.2 Generation of Object Diagrams

UMTG employs the Alloy analyzer to generate an object diagram which satisfies the path condition of a given scenario (Line 9 in Fig. 12). Test input values are later extracted from the generated object diagram (see Section 12).

The Alloy analyzer does not return a solution for unsatisfiable path conditions. These path conditions are usually associated with infeasible scenarios (i.e., scenarios which cannot be tested and verified by any set of possible input values). However, we may also observe unsatisfiable path conditions with feasible scenarios. These scenarios include an internal step that overrides the effects of the previous internal step (e.g., redefines the value of a variable). For instance, in Scenario B in Fig. 14, the constraints *C14* (i.e., $TemperatureLowError.allInstances() \rightarrow forAll(i|i.detected = true)$) and *C2* (i.e, $TemperatureError.allInstances() \rightarrow forAll(i|i.detected = false)$ ) cannot be satisfied in the same path condition since *TemperatureLowError* is a subclass of *TemperatureError*. The internal step with *C14* (Line 53 in Table 4) sets a temperature error to *detected* after the internal step with *C2* (Line 38 in Table 4) sets all the temperature errors to *undetected*. We identify such feasible scenarios having unsatisfiable path conditions by incrementally re-executing the Alloy analyzer.

When the Alloy analyzer does not solve a path condition, we automatically rerun it incrementally, by following the order of the steps in the scenario, until it identifies an unsatisfiable prefix of the path condition and returns the unsat core[2]. For instance, for Scenario B, UMTG runs the Alloy analyzer on (*C1*), (*C1* $\wedge$ *C2*), (*C1* $\wedge$ *C2* $\wedge$ *C3*), (*C1* $\wedge$ *C2* $\wedge$ *C3* $\wedge$ *C4*), (*C1* $\wedge$ *C2* $\wedge$ *C3* $\wedge$ *C4* $\wedge$ *C5*), and (*C1* $\wedge$ *C2* $\wedge$ *C3* $\wedge$ *C4* $\wedge$ *C5* $\wedge$ *C14*), respectively. For (*C1* $\wedge$ *C2* $\wedge$ *C3* $\wedge$ *C4* $\wedge$ *C5* $\wedge$ *C14*), the Alloy analyzer returns (*C2* $\wedge$ *C14*) as the unsat core.

If the last OCL constraint added to the path condition (e.g., *C14* in our example) belongs to an internal step $s_i$, we conclude that $s_i$ overrides the effects of previous internal steps. The other constraints in the unsat core can thus be safely removed from the path condition if they refer to the same attributes referred by this last OCL constraint. UMTG proceeds to incrementally solve the remaining constraints in the scenario. Otherwise, if the last added OCL constraint does not belong to an internal step or if the other constraints in the unsat core refer to different variables, the scenario is considered infeasible. For instance, both *C2* and *C14* belong to internal steps in Scenario B; *C14* is the last constraint added to the path condition before the unsat core is returned. Therefore, *C2* is removed from (*C1* $\wedge$ *C2* $\wedge$ *C3* $\wedge$ *C4* $\wedge$ *C5* $\wedge$ *C14*). We incrementally rerun the Alloy analyzer by adding the remaining constraints in Scenario B to (*C1* $\wedge$ *C3* $\wedge$ *C4* $\wedge$ *C5* $\wedge$ *C14*). Constraint solving proceeds until the scenario is deemed infeasible or the entire path condition excluding the removed OCL constraint(s) is solved. In this case, the Alloy analyzer returns an object diagram that satisfies the entire path condition for Scenario B shown in Fig. 14 excluding *C2*. The OCL constraints removed from the path condition during incremental constraint solving belong to internal steps that perform initial setups and thus do not have any effect on the identification of test inputs.

2. The unsat core is the minimal subset of OCL constraints that cannot be solved together [119]

## 12 Generation of Test Cases

UMTG uses the set of pairs $\langle scenario, objectDiagram \rangle$ to generate test cases (Step 10 in Fig. 1). It performs three activities: *identify input values*, *generate high-level operation descriptions*, and *generate test driver function calls* (see Fig. 16). Test driver function calls are tailored to the test case format in Table 2 but can be adapted, while following similar principles, to different test case formats for different test infrastructures and hardware.

TABLE 10
The test case automatically generated from Scenario A in Fig. 14.

| Line | Operation | Inputs/Expectations |
|---|---|---|
| 1 | *Input* | System.initialized = true |
| 2 | ResetPower | Time=INIT_TIME |
| 3 | *Input* | System.seatSensor.capacitance = 601 |
| 4 | SetBus | Channel=RELAY Capacitance=601 |
| 5 | *Input* | System.temperature = 20 |
| 6 | SetBus | Channel=RELAY Temperature = 20 |
| 7 | *Check* | An adult has been detected on the seat. |
| 8 | ReadAndCheckBus | D0=OCCUPIED<br>D1=OCCUPIED |
| 9 | *Check* | The occupant class for airbag control has been sent to AirbagControlUnit. The occupant class for seat belt reminder has been sent to SeatBeltControlUnit. |
| 10 | CheckAirbagPin | 0x010 |

Table 10 shows the test case automatically generated from Scenario A in Fig. 14. The test cases generated by UMTG follow the structure presented in Section 3. The odd lines contain high-level operation descriptions followed (even line numbers) by the name of the functions that should be executed by the test driver along with the corresponding input and expected output values. Lines 1, 3, 5, 7 and 9 are high-level operation descriptions; Lines 2, 4, 6, 8 and 10 are test driver function calls. The test case corresponds to the manually written test case in Table 2.

The input values are extracted from the object diagram generated from the path condition of the given scenario (Activity 1 in Fig. 16). To this end, UMTG looks for the attributes that appear both in the diagram and the path condition.

UMTG then generates the high-level operation descriptions, i.e., the *Input*, *Setup*, and *Check* operations (Activity 2 in Fig. 16). The *Input* and *Setup* operations are associated with the input values (Activities 2.1 - 2.3). For each *Input* and *InterruptingCondition* step in the scenario, we generate a test line including an *Input* operation. The *Input* operation is associated with a set of input values which belong to an attribute or an entity referenced in the use case step. For instance, in Activity 2.2, '*BodySense.seatSensor.capacitance = 601*' is selected because of the *Capacitance* entity. We generate a test line including the *Setup* operation for each attribute that appears in an OCL constraint belonging to *UseCaseStart* or *Condition* steps but not to *Input* and *InterruptingCondition* steps. For instance, in Activity 2.1, '*BodySense.initialized = true*' is selected because the attribute *initialized* is referenced in the use case precondition *The system has been initialized*.

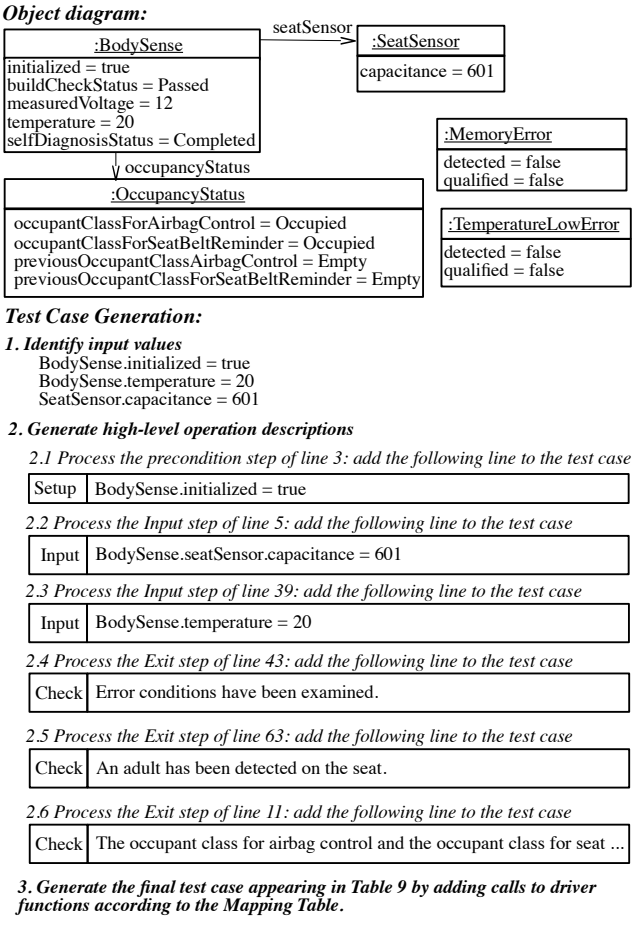


Fig. 16. Activities performed to generate the test case in Table 10.

TABLE 11
Part of the mapping table for $BodySense$.

| Pattern to match (Operation and Inputs) | | Result (Operation and Inputs) | |
|---|---|---|---|
| Input | BodySense.seatSensor.capacitance = (.*) | SetBus | Channel = RELAY Capacitance = \1 |
| Input | System.initialized = true | Reset Power | Time=INIT_TIME |
| Check | An adult has been detected on the seat. | ReadAnd CheckBus | D0=OCCUPIED D1=OCCUPIED |

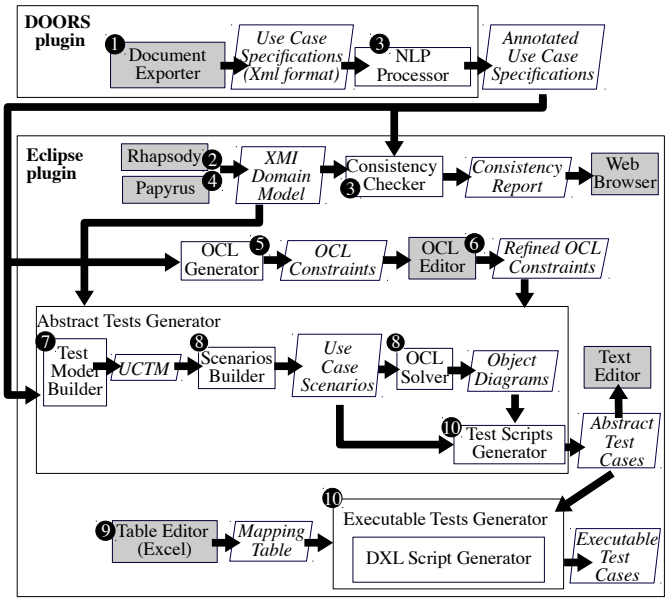


Fig. 17. UMTG toolset architecture. Grey boxes show third party components, while white boxes denote UMTG components. Black circles denote the steps in Fig. 1.

Finally, for the postcondition of each *Exit* step, we create a test line including the *Check* operation and the postcondition in NL (Activities 2.4 - 2.6). A *Check* operation is an abstract test oracle. The generated sequence of high-level operation descriptions is an *abstract test case*.

Finally, to generate the test driver function calls for the executable test case, UMTG processes the high-level operation descriptions by using the mapping table provided by the engineers (Activity 3 in Fig. 16). Table 11 shows part of the mapping table for $BodySense$. The first two columns provide the operation names and the regular expressions that match the high-level operation descriptions in the test case. The last two columns provide the test driver function calls to be added to the test case when the first two columns match the high-level operation descriptions. For instance, the *Input* operation and the expression *'BodySense.seatSensor.capacitance = (.*)'* in the first row of Table 11 matches the high-level operation description in Line 3 in Table 10, which leads to the generation of Line 4. We follow the Java regular expressions' syntax [120], which provides the grouping feature to extract char sequences from matching strings. For example, we use the grouping pattern *'(.*)'* to copy the values of the high-level operation descriptions, which are extracted from the object diagram, into the test driver function calls (e.g., value 601 in Activity 2.2).

Test oracles are the test driver function calls which are generated from the high-level operation descriptions with the *Check* operation (e.g., Lines 8 and 10 in Table 10).

To simplify the writing of mapping tables, engineers can create them iteratively and incrementally. They define mappings for the first test case and then introduce (and refine) regular expressions for the following test cases. Moreover, when the grouping feature of regular expressions is used to copy input values generated by the OCL solver into executable test cases, the same mapping table can be reused for test cases generated with different coverage criteria. Indeed, test cases derived with different coverage criteria differ only with respect to the values assigned to inputs and the order of operations. Finally, mapping tables can be shared across all the systems tested on a given test infrastructure.

## 13 Tool Support

To ease its industrial adoption, we implemented UMTG as a toolset that extends IBM Doors [121] and Eclipse IDE [122]. Additional information about the UMTG toolset, including executable files, download instructions and a screencast covering motivations, are available on the tool's website at https://sntsvv.github.io/UMTG/.

Fig. 17 shows the tool architecture. The main components are two plug-ins that extend IBM Doors and Eclipse IDE. They orchestrate the other components in Fig. 17.

The IBM Doors plug-in activates the UMTG steps for use case elicitation while the steps for test case generation are activated by the Eclipse plug-in. The toolset can also process use case specifications in plain text files to avoid any dependence on IBM Doors for requirements elicitation.

The Eclipse plug-in architecture enables us to rely on third party plug-ins. We rely on the *Eclipse Web Browser* to visualize the missing entities in the domain model, the *Eclipse OCL editor* [123] to edit the OCL constraints, the default *Eclipse Table Editor* (e.g., Microsoft Excel) to edit the mapping table, and the *Eclipse Project Explorer* to list the UMTG artefacts. To create the domain model, the user can use the IBM Rhapsody plug-in [124] or the Eclipse Papyrus plug-in [125]. UMTG requires that the domain model be saved in the XMI format.

The components in Fig. 17 automate Steps 3, 5, 7, 8, and 10 in Fig. 1. The *NLP Processor* contains the NL pipeline described in Section 7 to annotate the use case specifications as required by Steps 3, 5 and 7. It is based on the GATE workbench [109], an open source NLP framework. To load use case specifications from IBM Doors, we use *Doors Document Exporter*, an IBM Doors API that exports the Doors content as xml files.

The *Consistency Checker* implements Step 3 in Fig. 1. The *OCL Generator* implements Step 5; it relies on the *CNP SRL* tool, *VerbNet*, and *WordNet*. The *Abstract Tests Generator* includes the components *Test Model Builder*, *Scenarios Builder*, and *OCL Solver*, which implement Steps 7 and 8. The *OCL Solver* employs the Alloy analyzer [38] to generate object diagrams. The *Test Scripts Generator* partially implements Step 10 by generating abstract test cases, i.e., textual test scripts with input values and high-level operation descriptions. The *Test Scripts Generator* also filters out replicated test cases that might result from the execution of UMTG against multiple use cases that share a subset of scenarios (e.g., the use case scenarios for unsuccessful self diagnosis in *BodySense*). The *Eclipse Text Editor* is used to inspect the abstract test cases.

The *Executable Tests Generator* takes the mapping table and the abstract test cases as input, and generates the executable test cases as output (Step 10). It exploits the Door eXtension Language (DXL) to load the generated test cases into IBM Doors. The DXL is also used to automatically generate trace links between use cases and generated test cases, an important feature to support testing of safety critical systems.

Our toolset facilitates the adoption of UMTG into a software development process by supporting both the generation of test cases from scratch and the maintenance of the generated test cases in the presence of requirements changes (i.e., when use case specifications are updated). In both cases, engineers simply follow the UMTG steps in Fig. 1. In the case of requirements changes, the effort required by manual activities is minimal. More precisely, in Step 4 (*Refine Model*), engineers will likely revise only a limited number of missing entities because the update of requirements usually introduces a limited number of changes in the domain concepts. In Step 5, *Generate Constraints*, UMTG relies on cached results to generate OCL constraints for unchanged use case steps; engineers manually inspect only the OCL constraints generated from new or modified use case steps. In Step 6, *Specify Missing Constraints*, engineers specify only the OCL constraints that were not generated from the updated use case sentences. Finally, in Step 10, engineers rely on the mapping table derived for the previous version of the system. Consequently, they typically introduce only a limited number of mapping table entries.

## 14 Empirical Evaluation

In this section, we investigate, based on two industrial case studies in the automotive domain, the following Research Questions (RQs):

- ***RQ1. Does the approach automatically generate the correct OCL constraints?*** The generation of OCL constraints is a major building block of our approach. With this research question, we aim to measure the precision (i.e., the fraction of the generated OCL constraints that are correct) and recall (i.e., the fraction of the required OCL constraints that are correctly generated) of the OCL generation algorithm in Section 9.
- ***RQ2 How does the approach compare to manual test case generation based on expertise?*** In the second research question, we evaluate (i) if the approach generates test cases that exercise the same use case scenarios exercised by test cases manually written by experts and (ii) if it discovers additional, relevant scenarios not exercised by the manually written test cases.
- ***RQ3. How does the approach compare to manual test case generation in terms of resources?*** This research question aims to compare our approach with manual test case generation in terms of engineers' effort and time required to generate test cases.

As discussed in the introduction, our empirical evaluation focuses on requirements coverage instead of code coverage and fault detection. This is because the purpose of acceptance testing is not fault detection (i.e., verification) but validation. Earlier testing activities focus on verification. For example, ISO26262, the main international automotive standard, enforces statement, branch, and MC/DC coverage for unit testing only [3]. Therefore, analyzing the effectiveness of UMTG in terms of fault detection is out of the scope of this paper. Further, we do not compare UMTG with model-based approaches because their adoption in our context is not feasible (see Section 3).

### 14.1 Subjects of the Evaluation

The subjects of our evaluation are *BodySense* and Hands Off Detection (*HOD*), two embedded systems developed by our industry partner, IEE [21].

*BodySense* is a car seat occupant classification system that enables smart airbag deployment; it has been introduced in Section 3. *HOD* is an embedded system that checks if the driver has both hands on the steering wheel. It is used to enable and disable automatic braking with driver-assist features for safety. For instance, a car should not automatically brake if the driver does not have both his hands on the steering wheel. *HOD* measures the capacitance between a conductive layer in the steering wheel and the electrical ground in the car body or in the seat frame. It notifies the autonomous driving assistance system when the

driver does not have both his hands on the steering wheel. The complexity of these two systems lies mostly in how they deal with errors, e.g., sensor break-downs. Therefore, their test suites mostly concern the verification of system behavior in the presence of errors. The two case studies are representative of automotive embedded systems and share their typical characteristics, such as handling multiple types of errors and communicating results to other components based on error status and sensor data.

TABLE 12
Overview of the *BodySense* and *HOD* use case specifications.

| | Use cases | | | Use case flows | Steps |
|---|---|---|---|---|---|
| | All | Entry | Included by others | | |
| *BodySense* | 7 | 2 | 5 | 59 | 266 |
| *HOD* | 17 | 11 | 6 | 27 | 76 |

In IEE's business context, like in many others, use cases are central development artifacts which are used for communicating requirements among stakeholders, such as customers. Use case specifications provide a systematic description of system-actor interactions. UMTG requires use case specifications in the RUCM template. For *BodySense*, the specifications had been initially elicited by IEE engineers following the Cockburn template [87]. They were later rewritten according to the RUCM template. The *HOD* specifications, instead, were already written by IEE engineers using RUCM with the UMTG and RUCM tools [41].

Table 12 reports on the size of the use case specifications of the case studies. The *BodySense* and *HOD* specifications include 7 and 17 use cases, respectively. The number of use case flows is 59 for *BodySense* and 27 for *HOD*, which indicates that the systems must deal with multiple alternative executions (e.g., alternative flows in the case of errors). The difference in the number of use case flows is largely explained by *HOD* specifications being less complete than *BodySense* specifications. In general, we consider use case specifications to be *complete* when their condition steps fully characterize equivalence partitions for system inputs. For example, in *BodySense*, all error conditions are modeled by means of condition steps that explicitly characterize anomalous sensor data (e.g., sensor data not in a given range as in Line 42 of Table 4). Consequently, *BodySense* specifications include an alternative flow for every input partition leading to an error. In contrast, use case specifications are *incomplete* when such equivalence partitions are partially modeled. This is the case of *HOD* specifications, which do not include condition steps characterizing anomalous sensor data but only describe how the system should behave in the presence of a specific error or a class of errors (e.g., Line 7 of Table 4, which verifies if any error has been detected by the system). *HOD* specifications include less alternative flows than the *BodySense* ones. In this case, the descriptions of error conditions are given in other documents and the different error types appear in the domain model. To test the system under all the possible error conditions, engineers have to carefully read multiple specification documents including the domain model. The impact of the completeness of use case specifications on UMTG automated test generation is discussed in the following sections.

In Table 12, column *Entry* reports the number of use cases that are entry points for testing (i.e., the main use cases for deriving acceptance test cases). Column *Included by others* reports the number of use cases that are included by other use cases.

## 14.2 Results

This section discusses the results of our case studies, addressing in turn each of the research questions.

### *14.2.1 RQ1: Does the approach automatically generate the correct OCL constraints?*

To respond to RQ1, we used UMTG to automatically generate the OCL constraints required to drive test generation for *BodySense* and *HOD*. We compared the automatically generated OCL constraints with the OCL constraints manually written by the first author of the paper. The latter were written together with IEE engineers in dedicated workshops. The comparison was fully automated by means of string matching.

We computed precision (i.e., the fraction of automatically generated constraints that are correct) and recall (i.e., the fraction of the required constraints for test case generation that are correctly generated by UMTG). A sentence may appear in multiple use case steps. To fairly compute precision and recall, we considered these sentences only once.

Table 13 presents the results for RQ1. UMTG generated 70 and 34 constraints from 86 and 36 sentences for *BodySense* and *HOD*, respectively. The number of the processed sentences is lower than the number of the use case steps because UMTG does not need to generate constraints from input, output, and exit steps. Out of 70 and 34 constraints, 68 and 36 are correct, respectively. Precision is 0.97 for *BodySense* and 1.00 for *HOD*. These results are highly promising since almost each generated constraint is correct.

TABLE 13
Precision and Recall of the automated generation of OCL constraints.

| | Processed sentences | Generated constraints | | | Not Generated | Precision | Recall |
|---|---|---|---|---|---|---|---|
| | | All | Incorrect | Correct | | | |
| *BodySense* | 86 | 70 | 2 | 68 | 18 | 0.97 | 0.77 |
| *HOD* | 36 | 34 | 0 | 34 | 2 | 1.00 | 0.94 |

For certain sentences, the automated constraint generation fails (the columns *Not Generated* and *Incorrect* in Table 13) because of imprecise writing and inconsistent terminology. For example, the step '*The system VALIDATES THAT the temperature is valid*' does not specify what a valid temperature is (i.e., a valid temperature range). We also cannot automatically generate a constraint from the step '*The system VALIDATES THAT the occupancy status is valid*' because it is not straightforward to determine that '*is valid*' should be translated to `<> Error`. The noun `error` is not even an antonym of the adjective `valid`. As a result, especially for BodySense, recall is not as good as what we had hoped for. Recall is better for *HOD* than for *BodySense* since IEEE engineers wrote the *HOD* specifications after *BodySense*. They had more experience and were more careful in writing the *HOD* specifications.

To avoid imprecise sentences and inconsistent terminology, we improved the specifications and domain models. For example, we defined the notion *valid* as a derived attribute in the domain model of *BodySense*. Our changes for *BodySense* include eight additional derived attributes, the rewriting of five redundant and inconsistent sentences, the renaming of two attributes in the domain model, and the refactoring of five concepts to be modelled using entities (abstract classes in two cases) instead of attributes. For *HOD*, we introduced one derived attribute. This limited number of changes indicates that it is not difficult for engineers to improve their modeling practices.

TABLE 14
Precision and Recall with the Improved Specifications and Models.

| | Processed sentences | Generated constraints | | | Not Generated | Precision | Recall |
|---|---|---|---|---|---|---|---|
| | | All | Incorrect | Correct | | | |
| *BodySense* | 84 | 81 | 1 | 80 | 3 | 0.99 | 0.95 |
| *HOD* | 36 | 35 | 0 | 35 | 1 | 1.00 | 0.97 |
| **Overall** | 120 | 116 | 1 | 115 | 4 | 0.99 | 0.96 |

Table 14 presents the results obtained with the improved specifications and models, which in practice can be easily obtained by carefully comparing the specifications and domain models, a task our tool now supports. Unsurprisingly, UMTG achieves a particularly high recall, above 0.9 in both case studies. Overall, it achieves a precision of 0.99 and a recall of 0.96, another promising result indicating the potential usefulness of our constraint generation in practical settings. Among the sentences that are not correctly translated into OCL constraints, we identify (a) three sentences capturing internal behavior not useful for test input generation (e.g., '*The system loads the default calibration data*'), and (b) one sentence referring to concepts having names that are similar to other concepts in the domain model, which, in turn, leads to the identification of the wrong entity for the constraint.

#### *14.2.2 RQ2: How does the approach compare to manual test case generation based on expertise?*

To respond to RQ2, we compared the test cases automatically generated by UMTG (hereafter *UMTG test cases*) with the test cases manually written, based on extensive expertise, by IEE engineers (hereafter *manual test cases*) for *BodySense* and *HOD*. We focused on use case scenarios and test inputs used to exercise the scenarios.

To compare the test cases based on use case scenarios, we inspected all the manual test cases and mapped them to the scenarios in the use case test models. To compare the test cases based on test inputs, we identified equivalence classes for each input domain and all the input partitions (i.e., the combination of equivalence classes) that might be considered for each scenario. We kept track of the input partitions covered by UMTG and manual test cases. We use the term *input scenario*, different than *use case scenario*, to refer to a use case scenario being exercised with a specific input partition.

Because of the nature of the practical problem we address, we have to rely on NLP. Such techniques are heuristics trained on large text corpora and are therefore heuristics. Therefore no proof of soundness is possible. We can, however, empirically assess soundness and this is what we do in our two representative industrial case studies. We manually verified that the sequences of instructions in the test cases satisfying the branch and def-use coverage criteria were consistent with the use case specifications. Furthermore, for the test cases generated with the subtype coverage criterion, we verified that the covered input partitions (i.e., the types of errors being triggered by test execution) were consistent with the use case specifications. In our analysis, we did not discover any test case being erroneously generated in the presence of correct OCL constraints, thus providing evidence that the approach is sound.

TABLE 15
Number of test cases in our evaluation.

| Case study | Manual test cases | UMTG test cases | | |
|---|---|---|---|---|
| | | Branch | Def-use | Subtype |
| *BodySense* | 134 | 63 | 97 | 1285 |
| *HOD* | 50 | 21 | 22 | 63 |

TABLE 16
Use case scenarios in the manual test cases that are also exercised by the UMTG test cases.

| Case study | Exercised by the manual test cases | Exercised also by the UMTG test cases | | |
|---|---|---|---|---|
| | | Branch | Def-use | Subtype |
| *BodySense* | 64 | 43 (67.19%) | 64 (100%) | 64 (100%) |
| *HOD* | 21 | 21 (100%) | 21 (100%) | 21 (100%) |

Table 15 presents the number of test cases in the test suites considered in our evaluation (i.e., the manual test suite developed by IEE and the test suites generated with three UMTG coverage strategies). An in-depth discussion concerning the size and the complementarity of the generated test suites is reported at the end of this section.

Table 16 presents the number of use case scenarios exercised by manual test cases and the percentage of these scenarios exercised by UMTG test cases. Except for branch coverage in *BodySense*, the UMTG test cases exercise all the scenarios exercised by the manual test cases. The UMTG test cases for branch coverage do not exercise all the scenarios covering the detection of each type of error in the presence of qualified errors given that branch coverage is satisfied when at least one test case covers the branch for error qualification. Note that the UMTG test cases cover all the branches covered by the manual test cases.

Table 17 shows the number of input scenarios exercised by manual test cases and the percentage of these input scenarios exercised by UMTG test cases. UMTG fares worse with branch coverage because the test cases do not exercise scenarios with all input partitions. With def-use coverage, UMTG achieves a much better result for *BodySense*, i.e., 97.76% of the manual scenarios being exercised. This happens because the use case specifications of *BodySense* are complete and capture all conditions that lead to error detection. Therefore, the use case scenarios that can be covered only by one single input partition make def-use coverage sufficient to cover all the input scenarios. A different result is achieved for *HOD* with def-use coverage. The *HOD* specifications are incomplete, which leads to multiple input partitions per use case scenario. Therefore, in *HOD*, subtype coverage is necessary to exercise the input scenarios covered

TABLE 17
Input scenarios in the manual test cases that are also exercised by the UMTG test cases.

| Case study | Exercised by the manual test cases | Exercised also by the UMTG test cases | | |
|---|---|---|---|---|
| | | Branch | Def-use | Subtype |
| *BodySense* | 134 | 96 (71.64%) | 131 (97.76%) | 134 (100%) |
| *HOD* | 56 | 21 (37.50%) | 21 (37.50%) | 50 (100%) |

by the manual test cases.

TABLE 18
Use case scenarios in the UMTG test cases that are also exercised by the manual test cases.

| Case study | Use case scenarios covered by the generated test cases | | | | | |
|---|---|---|---|---|---|---|
| | Branch | | Def-use | | Subtype | |
| | UMTG | Exercised by manual | UMTG | Exercised by manual | UMTG | Exercised by manual |
| *BodySense* | 63 | 43 (68.25%) | 97 | 64 (65.98%) | 97 | 64 (65.98%) |
| *HOD* | 22 | 21 (94.45%) | 22 | 21 (95.45%) | 22 | 21 (95.45%) |

TABLE 19
Input scenarios in the UMTG test cases that are also exercised by the manual test cases.

| Case study | Input scenarios covered by the generated test cases | | | | | |
|---|---|---|---|---|---|---|
| | Branch | | Def-use | | Subtype | |
| | UMTG | Exercised by manual | UMTG | Exercised by manual | UMTG | Exercised by manual |
| *BodySense* | 63 | 43 (68.25%) | 97 | 64 (65.98%) | 867 | 74 (8.54%) |
| *HOD* | 22 | 18 (81.82%) | 22 | 18 (81.82%) | 71 | 56 (78.87%) |

Table 18 presents the number of use case scenarios exercised by UMTG test cases and the percentage of these scenarios exercised by manual test cases. UMTG test cases exercise more use case scenarios than manual test cases. For example, in *BodySense*, only 68% of the scenarios for branch coverage are exercised by manual test cases. With branch coverage, UMTG systematically covers all the scenarios derived from bounded and global alternative flows. It is difficult to manually identify these scenarios because engineers need to create a scenario for each reference flow step which a bounded (or global) flow refers to. Therefore, engineers tend not to test all the scenarios that can be derived from bounded and global alternative flows. With def-use coverage, UMTG systematically covers all def-use pairs. In our case studies, the def-use pairs consist of internal steps reporting the presence of specific errors and condition steps verifying the presence of these errors. In turn, UMTG ensures that the systems are tested for each error in all possible combinations of condition steps verifying the presence of the error (e.g., Scenarios B and C in Fig. 14). Such systematic combination coverage is hard to achieve for engineers. By definition, with the subtype and def-use coverage criteria, UMTG generates the same use case scenarios.

Table 19 shows the number of input scenarios exercised by UMTG test cases and the percentage of these input scenarios exercised by manual test cases. UMTG test cases exercise more input scenarios than manual test cases. For the branch and def-use coverage criteria, the numbers of input and use case scenarios exercised by UMTG test cases are the same (see Tables 18 and 19). With subtype coverage, UMTG systematically covers each condition step for all possible subtypes (e.g., for all error types in our case studies). In our case studies, to speed up manual test case generation, engineers do not consider all error types in condition steps.

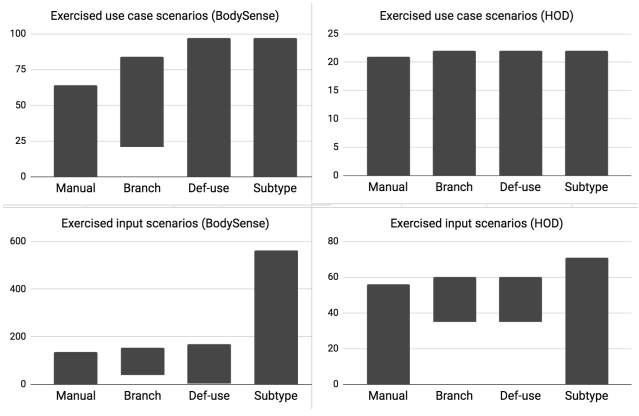


Fig. 18. Use case and input scenarios exercised by the UMTG and manual test suites.

To summarize our findings, Fig. 18 presents four bar charts comparing all the use case and input scenarios exercised by the manual and UMTG test suites. In the charts, each data point on the vertical axis represents a use case or input scenario exercised by one or more test suites. When a scenario is exercised by a test suite, a blue horizontal line is drawn in the vertical axis. A blue bar indicates a group of test cases in a test suite. For example, the *BodySense* use case scenario matching data point 50 is exercised by four test suites, while the use case scenario for data point 10 is not exercised by the branch coverage test suite. We sort the data points to simplify the visual comparison of their coverage.

Branch coverage is the least effective one because it leads to test suites that do not exercise all the scenarios exercised by the manual test suite. For example, in *BodySense*, the branch coverage test suite exercises 20 use case scenarios that are not exercised by the manual test suite (the data points between 65 and 84 are covered only by the branch coverage test suite) but it does not exercise 48 use case scenarios in the manual test suite (the data points below 49 are not covered by the branch coverage test suite). Unsurprisingly, def-use coverage subsumes branch coverage. Finally, the chart shows that subtype coverage is the only criterion that exercises all the use case and input scenarios in the manual test suite in addition to input scenarios not exercised by the manual test suite. Unfortunately, the associated high number of test cases increases the cost of testing. On the other hand, subtype coverage is necessary only when specifications are incomplete, i.e., the *HOD* case where def-use coverage misses a lot of input scenarios. Indeed, in *BodySense*, the def-use coverage test suite exercises all the input scenarios covered by the manual test suite except three input scenarios. These three input scenarios exercise a use case scenario that concerns a change in the occupancy status of the seat in the presence of specific errors being both detected and qualified. However, UMTG covers the same use case scenario with other types of errors not considered in the manual test suite. There is no specific reason for test-

TABLE 20
Information to be manually provided when relying on UMTG and manual test case generation.

| Case study | Manual testing | | UMTG | | |
|---|---|---|---|---|---|
| | Sentences (words) in scenarios descriptions | Executable instructions | Reviewed constraints | Manually written OCLs | Lines in mapping table |
| *BodySense* | 2,476 (18,716) | 8,886 | 71 | 20 | 67 |
| *HOD* | 408 (3,890) | 1,229 | 35 | 2 | 51 |

TABLE 21
UMTG test generation time and number of calls to constraint solver.

| Case study | Max test generation time for all the scenarios (minutes) | | | Calls to constraint solver (total for all the scenarios) | | |
|---|---|---|---|---|---|---|
| | Branch | Def-use | Subtype | Branch | Def-use | Subtype |
| *BodySense* | 330 | 720 | 7,200 | 908 | 2,762 | 23,913 |
| *HOD* | 2 | 2 | 4 | 43 | 46 | 154 |

ing the above-mentioned use case scenario with a specific subset of errors rather than another. Therefore, we conclude that def-use coverage is sufficient to generate a test suite for *BodySense* that is as exhaustive as the manual one.

Last, we observe that, for branch coverage (in *HOD*) and def-use coverage (in *BodySense*), UMTG exercises more use case scenarios than manual test cases, though it generates less test cases (see Table 15). This is due to the fact that in the case of manual test cases, engineers, to simplify the identification of test inputs, often derive test cases that focus only on the scenarios in the included use cases. UMTG, instead, generates test cases that start with entry point use cases and terminate in abort or exit steps of entry point use cases; consequently, each UMTG test case may cover multiple subsequences of use case steps belonging to distinct included use cases. For this reason, the same UMTG test case may cover sequences of use case steps exercised by distinct manual test cases in addition to sequences of use case steps not covered by any manual test case. This is also reflected in the results obtained for the coverage of input scenarios. Indeed, the same UMTG test case may exercise multiple input scenarios covered by distinct manual test cases; this is the reason why, for example, the 63 input scenarios in the test cases generated by UMTG with the branch coverage criterion (see Table 19) exercise 96 input scenarios in the manual test cases (see Table 17). More precisely, we observe that engineers implemented multiple test cases to separately exercise distinct subsequences of use case steps for a specific error condition. Instead, a single UMTG test case exercises all such subsequences together against the same error condition. Since the input scenarios exercised by manual test cases cover subsequences of steps exercised by UMTG test cases, in Table 17, multiple instances of the former may be covered by a single instance of the latter, with the same input partitions. Instead, in Table 19, an input scenario exercised by a UMTG test case is counted as covered by manual test cases when all the use case steps of the scenario are covered by one or more manual test cases, exercising all steps or a distinct subset, with the same input partitions. Finally, we report that, in general, we generate a distinct test case for each input scenario; however, this is not the case for the manual test cases in HOD where the same test case covers multiple input partitions after resetting the system.

#### *14.2.3 RQ3: How does the approach compare to manual test case generation in terms of resources?*

RQ3 is concerned with the effort and time required to generate test cases. IEE engineers, for each manual test case, write a scenario including executable instructions and provide a descriptive comment in each executable instruction. In UMTG, engineers review automatically generated OCL constraints, write missing constraints, and define a mapping table for a given test infrastructure, reusable across systems. We do not consider the effort required to derive use case specifications and domain models since they are typically produced during requirements analysis for other purposes [22].

Table 20 presents the data characterizing the manual activities in our case studies. In manual test case generation, IEE engineers had to write a large number of sentences to document the test cases, i.e., 2,476 and 408 for *BodySense* and *HOD*, respectively. The test cases of the two systems include a total of 8,886 and 1,229 executable instructions and code comments. Thus, manual test case generation is expensive and generally takes several days (at least five working days for each case study).

In UMTG, engineers reviewed 71 and 35 OCL constraints for *BodySense* and *HOD*, respectively. The review was fast (on average less than two minutes per constraint) since the constraints are short and follow a well-defined pattern. The number of constraints written by engineers is low, i.e., 20 for *BodySense* and 2 for *HOD*. The mapping tables for *BodySense* and *HOD* include 67 and 51 lines, respectively. The time taken to write the mapping tables is limited, i.e., 135 minutes for *BodySense* and 100 minutes for *HOD*. Mapping table generation is much faster than manual test case generation which may last days. In the case studies, the mapping tables for the three code coverage strategies are the same. When use case specifications are complete, the specific values that trigger error conditions (e.g., *BodySense.temperature = 100*) are generated by the constraint solver and processed in the mapping table by means of the grouping feature of regular expressions (e.g., Line 6 of Table 10). When use case specifications are incomplete, test case generation identifies the error condition that should hold in a scenario (e.g., *TemperatureHighError.isDetected = true*) and the mapping table is used to translate such conditions into concrete values (i.e., *TemperatureHighError.isDetected = true* is mapped to *SetBus: Channel=RELAY Temperature = 20*). When specifications are incomplete, mapping tables become more complex. Incomplete use case specifications force engineers to encode system-specific, functional parameters in mapping tables, a practice that may limit the reuse of mapping tables across systems. Therefore, we recommend that engineers produce complete use case specifications.

Table 21 presents the time, in minutes, that UMTG took to automatically generate test cases for *BodySense* and *HOD*. Since test cases for different scenarios can be generated in parallel, we report the time of the longest test case generation. To generate test cases, UMTG needed more time

for *BodySense* because its use case specifications have more alternative flows which, in turn, lead to larger use case test models to be traversed and more path conditions to be solved. With branch and def-use coverages, it took from 1 minutes to 12 hours, which is acceptable since it is fully automated and still faster than manual test case generation. Manual test case generation took between two and five working days for the same case studies. With subtype coverage, UMTG needed time up to 120 hours (5 days). Technological improvements such as parallel execution of constraint solving (we have 23,913 invocations to the constraint solver) may drastically reduce test case generation time. Subtype coverage is built on top of def-use coverage. Engineers can execute test cases generated based on def-use coverage and start repairing some of the faults to be detected by subtype coverage. This can significantly speed up acceptance testing.

The OCL constraint generation does not practically impact on the test case generation time. It takes 310 seconds for *BodySense* and 110 seconds for *HOD*. The OCL generation time mostly depends on the SRL execution and the network communication with the CNP server, which take between 1 and 4 seconds for each use case sentence.

Based on our observations above, we are confident that, in practice, UMTG requires less effort and time than manual test case generation. It can be easily integrated into the development process and provides better guarantees for achieving use case and input scenario coverage.

### 14.3 Discussion

The modeling effort required for testing a software system with UMTG is minimized thanks to the automated generation of OCL constraints. Our results show that UMTG automatically and correctly generated 95% of the constraints required for testing the two systems in our evaluation.

Among all the coverage strategies in UMTG, def-use coverage performs the best. Indeed, it enables the generation of test cases that exercise all the use case scenarios covered by the manual test suites; in the presence of complete specifications, it enables engineers to automatically cover all the input scenarios considered in the manual test suites.

To cover an adequate set of input partitions (at least the ones covered by manual test case generation), engineers should select the coverage criterion based on the completeness of specifications. Def-use coverage is sufficient when specifications are complete (i.e., when they capture all input partitions by means of condition steps) while subtype coverage is needed when specifications are incomplete (i.e., when they do not explicitly capture input partitions by means of condition steps). Based on our experience, test case generation for these two cases can be performed overnight, which makes UMTG appealing in industrial contexts.

UMTG leads to significantly less test generation time than manual test case generation. The latter also consumes substantial human effort (days) whereas the former mostly requires computation time along with hours of human effort, including writing constraints and mapping tables. This is in practice an important distinction as computation time is much more readily available than human skills. Also, in the presence of specifications that are often updated (e.g., to reconfigure the system because it is part of a product line), automated test case generation relieves engineers from the burden of the manual identification of use case scenarios that need to be tested with newly implemented test cases.

### 14.4 Threats to Validity

The main threat to validity in our study relates to generalizability, a common issue with industrial case studies. To deal with this threat, we consider two representative automotive embedded systems implementing very different functionalities, sold to different customers, with specifications written by different sets of people following different requirements analysis practices (*BodySense* specifications are complete, while *HOD* specifications are incomplete).

In addition, in Section 9, we show that the OCL pattern handled by UMTG is expressive enough for embedded systems, while the requirements specification format processed by UMTG (use case specifications) is common in industrial settings that require precise requirements elicitation shared by multiple stakeholders.

## 15 Conclusion

In this paper, we introduced UMTG, an approach to automate acceptance testing, with a focus on embedded systems. It generates test data and executable, system-level, test cases for the purpose of validating conformance with requirements, an activity usually referred to as acceptance testing. We rely on use case specifications augmented with a domain model. Our motivation is to achieve automated test case generation by largely relying on common practices to document requirements for communication purposes among stakeholders in embedded system domains.

To enable the automatic identification of use case scenarios and test inputs, we combine NLP and constraint solving. To extract behavioral information from use case specifications by means of NLP, we rely upon a restricted use case modeling method called RUCM. We use an advanced NLP solution (i.e., semantic role labeling) to automatically generate OCL constraints that capture the pre and postconditions of use case steps. We designed an algorithm that identifies use case scenarios based on the branch, def-use, and subtype coverage criteria. It builds feasible path conditions that capture OCL constraints under which the alternative flows in each scenario are executed. Test inputs are determined by solving the path conditions with the Alloy analyzer.

Two industrial case studies show that UMTG effectively generates acceptance test cases for automotive sensor systems. UMTG can automatically and correctly generate 96% of the OCL constraints for test case generation. The precision of the OCL generation process is very high: 99% of the generated constraints are correct. The coverage criteria implemented in UMTG enable the identification of use case scenarios missed by engineers, thus highlighting its usefulness in safety domains.

Our results show that the subtype coverage criterion, when specifications are incomplete, and the def-use coverage criterion, in general, generate test cases that cover all the scenarios and input partitions exercised by the test cases written by experts. This is achieved by automatically

generated test suites that have the same size as the manually implemented ones, thus not affecting testing cost. Our experience indicates that requirements modeling, mostly limited in UMTG to RUCM use case specifications and domain modeling, is feasible in an industrial context. Furthermore, manual test case generation requires significantly more effort than applying UMTG, since use case specifications and domain models are usually employed for other purposes as well.

Future work includes the extension of UMTG to deal with different types of testing problems and the identification of solutions to further the scalability of the strategy due to constraint solving for very large systems. We are currently working on UMTG-inspired approaches for security testing of Web systems [33], [34]. We also aim to address scalability issues by evaluating the feasibility of adopting alternative solving approaches, including SMT solvers [126], answer set programming [127], higher-order relational constraint solving [128], and the combination of constraint solving and search-based optimization [95], [129].

## Acknowledgments

We gratefully acknowledge funding from FNR and IEE S.A. Luxembourg, the grant FNR/P10/03, the Canada Research Chair program, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).

## References

[1] I. S. Association *et al.*, "Systems and software engineering—vocabulary iso/iec/ieee 24765: 2017," pp. 1–536, 2017.

[2] Radio Technical Commission for Aeronautics-RTCA, "DO-178C, software considerations in airborne systems and equipmen certification," 2011.

[3] International Organization for Standardization-ISO, "ISO-26262: Road vehicles – functional safety."

[4] B. Anda, K. Hansen, and G. Sand, "An investigation of use case quality in a large safety-critical software development project," *Information and Software Technology*, vol. 51, no. 12, pp. 1699–1711, 2009.

[5] C. Nebut, F. Fleurey, Y. Le-Traon, and J.-M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[6] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, "Extracting domain models from natural-language requirements: Approach and industrial evaluation," in *MODELS'16*, 2016, pp. 250–260.

[7] L. Provenzano and K. Hänninen, "Specifying software requirements for safety-critical railway systems: An experience report," in *REFSQ'17*, 2017, pp. 363–369.

[8] M. J. Escalona, J. J. Gutierrez, M. Mejías, G. Aragón, I. Ramos, J. Torres, and F. J. Domínguez, "An overview on test generation from functional requirements," *Journal of Systems and Software*, vol. 84, no. 8, pp. 1379–1393, 2011.

[9] M. Shirole and R. Kumar, "UML behavioral model based test case generation: a survey," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, pp. 1–13, 2013.

[10] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from UML activity diagram based on gray-box method," in *APSEC'04*, 2004, pp. 284–291.

[11] J. Ryser and M. Glinz, "A scenario-based approach to validating and testing software systems using statecharts." in *ICSSEA'99*, 1999.

[12] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the transition from use case models to analysis models: Approach and experiments," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, 2013.

[13] T. Yue, S. Ali, and L. Briand, "Automated transition from use cases to UML state machines to support state-based testing," in *ECMFA'11*, 2011, pp. 115–131.

[14] T. Yue, L. Briand, and Y. Labiche, "An automated approach to transform use cases into activity diagrams," in *ECMFA'10*, 2010, pp. 337–353.

[15] M. Zhang, T. Yue, S. Ali, H. Zhang, and J. Wu, "A systematic approach to automatically derive test cases from use cases specified in restricted natural languages," in *SAM'14*, 2014, pp. 142–157.

[16] E. Sarmiento, J. C. Leite, E. Almentero, and G. S. Alzamora, "Test scenario generation from natural language requirements descriptions based on petri-nets," *Electronic Notes in Theoretical Computer Science*, vol. 329, pp. 123–148, 2016.

[17] E. Sarmiento, J. C. S. d. P. Leite, and E. Almentero, "C&L: Generating model based test cases from natural language requirements descriptions," in *RET'14*, 2014, pp. 32–38.

[18] J. Gutiérrez, M. J. Escalona, M. Mejías, and J. Torres, "A case study for generating test cases from use cases," in *RCIS'08*, 2008, pp. 83–96.

[19] A. L. L. de Figueiredo, W. L. Andrade, and P. D. L. Machado, "Generating interaction test cases for mobile phone systems from use case specifications," *SIGSOFT Software Engineering Notes*, vol. 31, no. 6, pp. 1–10, 2006.

[20] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "NAT2TEST SCR: Test case generation from natural language requirements based on SCR specifications," *Science of Computer Programming*, vol. 95, pp. 275–297, 2014.

[21] "IEE (International Electronics & Engineering) S.A." http://www.iee.lu.

[22] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall Professional, 2002.

[23] J. Zhou, Y. Lu, K. Lundqvist, H. Lonn, D. Karlsson, and B. Liwang, "Towards feature-oriented requirements validation for automotive systems," in *RE'14*, 2014, pp. 428–436.

[24] T. Yue, S. Ali, and M. Zhang, "RTCM: A natural language based, automated, and practical test case generation framework," in *ISSTA'15*, 2015, pp. 397–408.

[25] I. Hajri, A. Goknil, L. C. Briand, and T. Stephany, "Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach," in *MODELS'15*, 2015, pp. 338–347.

[26] ——, "PUMConf: a tool to configure product specific use case and domain models in a product line," in *ESEC/FSE'16*, 2016, pp. 1008–1012.

[27] ——, "Configuring use case models in product families," *Software & Systems Modeling*, vol. 17, no. 3, pp. 939–971, 2018.

[28] ——, "Change impact analysis for evolving configuration decisions in product line use case models," *Journal of Systems and Software*, vol. 139, pp. 211–237, 2018.

[29] ——, "Incremental reconfiguration of product specific use case models for evolving configuration decisions," in *REFSQ'17*, 2017, pp. 3–21.

[30] I. Hajri, A. Goknil, and L. C. Briand, "A change management approach in product lines for use case-driven development and testing," in *Poster and Tool Track at REFSQ'17*, 2017.

[31] P. X. Mai, A. Goknil, L. K. Shar, F. Pastore, L. C. Briand, and S. Shaame, "Modeling security and privacy requirements: a use case-driven approach," *Information and Software Technology*, vol. 100, pp. 165–182, 2018.

[32] I. Hajri, A. Goknil, F. Pastore, and L. C. Briand, "Automating test case classification and prioritization for use case-driven testing in product lines," *arXiv preprint arXiv:1905.11699*, 2019.

[33] P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "A natural language programming approach for requirements-based security testing," in *ISSRE'18*, 2018, pp. 58–69.

[34] ——, "MCP: A security testing tool driven by requirements," in *ICSE'19*, 2019, pp. 55–58.

[35] M. Zhang, T. Yue, S. Ali, B. Selic, O. Okariz, R. Norgre, and K. Intxausti, "Specifying uncertainty in use case models," *Journal of Systems and Software*, vol. 144, pp. 573–603, 2018.

[36] "The Object Constraint Language (OCL)," http://www.omg.org/spec/OCL/.

[37] V. Punyakanok, D. Roth, and W. Yih, "The importance of syntactic parsing and inference in semantic role labeling," *Computational Linguistics*, vol. 34, no. 2, 2008.

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

[39] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *ISSTA'15*, 2015, pp. 385–396.

[40] C. Wang, F. Pastore, and L. C. Briand, "Automated generation of constraints from use case specifications to support system testing," in *ICST'18*, 2018, pp. 23–33.

[41] C. Wang, F. Pastore, A. Goknil, L. C. Briand, and M. Z. Z. Iqbal, "UMTG: a toolset to automatically generate system test cases from use case specifications," in *ESEC/SIGSOFT FSE'15*, 2015, pp. 942–945.

[42] A. Pretschner, "Model-based testing," in *ICSE'05*, 2005, pp. 722–723.

[43] W. Li, F. Le Gall, and N. Spaseski, "A survey on model-based testing tools for test case generation," in *TMPA'17*. Springer, 2017, pp. 77–89.

[44] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[45] D. Jurafsky and J. H. Martin, *Speech and Language Processing (3rd ed.)*, 3rd ed. Prentice Hall, 2017, draft Online: https://web.stanford.edu/~jurafsky/slp3/.

[46] B. Zhang, E. Hill, and J. Clause, "Automatically generating test templates from test names (n)," in *ASE'15*, 2015, pp. 506–511.

[47] A. Sinha, A. Paradkar, P. Kumanan, and B. Boguraev, "A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases," in *DSN'09*, 2009, pp. 327–336.

[48] I. S. Bajwa, B. Bordbar, and M. G. Lee, "OCL constraints generation from natural language specification," in *EDOC'10*, 2010, pp. 204–213.

[49] I. S. Bajwa, B. Bordbar, K. Anastasakis, and M. Lee, "On a chain of transformations for generating alloy from NL constraints," in *ICDIM'12*, 2012, pp. 93–98.

[50] C. Arora, M. Sabetzadeh, A. Goknil, L. C. Briand, and F. Zimmer, "Change impact analysis for natural language requirements: An nlp approach," in *RE'15*, 2015, pp. 6–15.

[51] ——, "NARCIA: An automated tool for change impact analysis in natural language requirements," in *ESEC/SIGSOFT FSE'15*, 2015, pp. 962–965.

[52] Carnegie Mellon University, "SEMAFOR," 2017. [Online]. Available: http://www.cs.cmu.edu/~ark/SEMAFOR/

[53] University of Saarland, "Shalmaneser," 2017. [Online]. Available: http://www.coli.uni-saarland.de/projects/salsa/shal/

[54] University of Illinois, "CogComp NLP Pipeline," 2017. [Online]. Available: https://github.com/CogComp/cogcomp-nlp/tree/master/pipeline

[55] D. Das, D. Chen, A. F. T. Martins, N. Schneider, and N. A. Smith, "Frame-semantic parsing," *Computational Linguistics*, vol. 40, no. 1, pp. 9–56, 2014.

[56] M. Palmer, D. Gildea, and P. Kingsbury, "The proposition bank: An annotated corpus of semantic roles," *Computational Linguistics*, vol. 31, no. 1, pp. 71–106, 2005.

[57] A. Meyers, R. Reeves, C. Macleod, R. Szekely, V. Zielinska, B. Young, and R. Grishman, "The NomBank project: An interim report," in *HLT-NAACL'04*, 2004, pp. 24–31.

[58] M. Gerber, J. Chai, and A. Meyers, "The role of implicit argumentation in nominal srl," in *NAACL'09*, 2009, pp. 146–154.

[59] K. Kipper, H. T. Dang, and M. Palmer, "Class-based construction of a verb lexicon," in *AAAI'00*, 2000, pp. 691–696.

[60] University of Colorado, "VerbNet: up to date list of verbnet classes." 2017. [Online]. Available: http://verbs.colorado.edu/verb-index/vn/class-h.php

[61] M. Palmer, "SemLink: Linking PropBank, VerbNet and FrameNet," in *GL'09*, 2009, pp. 9–15.

[62] G. A. Miller, "WordNet: A lexical database for english," *ACM Communications*, vol. 38, no. 11, pp. 39–41, 1995.

[63] A. Bandyopadhyay and S. Ghosh, "Test input generation using UML sequence and state machines models," in *ICST'09*, 2009, pp. 121–130.

[64] L. C. Briand and Y. Labiche, "A UML-based approach to system testing," *Software & Systems Modeling*, vol. 1, no. 1, pp. 10–42, 2002.

[65] J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in *UML'99*, 1999, pp. 416–429.

[66] A. Abdurazik and J. Offutt, "Using UML collaboration diagrams for static checking and test generation," in *UML'00*, 2000, pp. 383–395.

[67] B. Hasling, H. Goetz, and K. Beetz, "Model based testing of system requirements using UML use case models," in *ICST'08*, 2008, pp. 367–376.

[68] A. Nayak and D. Samanta, "Synthesis of test scenarios using UML activity diagrams," *Software & Systems Modeling*, vol. 10, pp. 63–89, 2011.

[69] P. Samuel and R. Mall, "Slicing-based test case generation from UML activity diagrams," *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 6, pp. 1–14, 2009.

[70] M. Sarma and R. Mall, "Automatic generation of test specifications for coverage of system state transitions," *Information and Software Technology*, vol. 51, no. 2, pp. 418–432, 2009.

[71] Y. G. Kim, H. S. Hong, D.-H. Bae, and S. D. Cha, "Test cases generation from UML state diagrams," *IEE Proceedings-Software*, vol. 146, no. 4, pp. 187–192, 1999.

[72] P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini, "TESTOR: Deriving test sequences from model-based specifications," in *CBSE'05*, 2005, pp. 267–282.

[73] F. Basanieri, A. Bertolino, and E. Marchetti, "The cow_suite approach to planning and deriving test suites in UML projects," in *UML'02*, 2002, pp. 383–397.

[74] J. Gutiérrez, M. Escalona, and M. Mejías, "A model-driven approach for functional test case generation," *Journal of Systems and Software*, vol. 109, pp. 214–228, 2015.

[75] J. J. Gutiérrez, C. Nebut, M. J. Escalona, M. Mejías, and I. M. Ramos, "Visualization of use cases through automatically generated activity diagrams," in *MODELS'08*, 2008, pp. 83–96.

[76] T. Yue, L. C. Briand, and Y. Labiche, "aToucan: an automated framework to derive UML analysis models from use case models," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, 2015.

[77] Z. Ding, M. Jiang, and M. Zhou, "Generating petri net-based behavioral models from textual use cases and application in railway networks," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 12, pp. 3330–3343, 2016.

[78] P. Frohlich and J. Link, "Automated test case generation from dynamic models," in *ECOOP'00*, 2000, pp. 472–491.

[79] V. A. de Santiago Junior and N. L. Vijaykumar, "Generating model-based test cases from natural language requirements for space application software," *Software Quality Journal*, vol. 20, no. 1, pp. 77–143, 2012.

[80] M. Riebisch, I. Philippow, and M. Götze, "UML-based statistical test case generation," in *NODe'02*, 2002, pp. 394–411.

[81] M. Katara and A. Kervinen, "Making model-based testing more agile: a use case driven approach," in *HVC'06*, 2006, pp. 219–234.

[82] S. Nogueira, A. Sampaio, and A. Mota, "Test generation from state based use case models," *Formal Aspects of Computing*, vol. 26, no. 3, pp. 441–490, May 2014.

[83] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *TOOLS'12*, 2012, pp. 269–287.

[84] "The stanford parser: A statistical parser," https://nlp.stanford.edu/software/lex-parser.shtml.

[85] J. Hartmann, M. Vieira, H. Foster, and A. Ruder, "A UML-based approach to system testing," *Innovations in Systems and Software Engineering*, vol. 1, no. 1, pp. 12–24, 2005.

[86] N. Kesserwan, R. Dssouli, J. Bentahar, B. Stepien, and P. Labrèche, "From use case maps to executable test procedures: a scenario-based approach," *Software & Systems Modeling*, 2017.

[87] A. Cockburn, *Writing Effective Use Cases*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[88] R. J. Buhr, "Use case maps as architectural entities for complex systems," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, pp. 1131–1155, 1998.

[89] A. Sinha, S. M. S. Jr., and A. Paradkar, "Text2Test: Automated inspection of natural language use cases," in *ICST'10*, 2010, pp. 155–164.

[90] P. Stocks and D. Carrington, "A framework for specification-based testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777–793, 1996.

[91] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 353–363, 1994.

[92] A. J. Offutt and S. Liu, "Generating test data from SOFL specifications," *Journal of Systems and Software*, vol. 49, no. 1, pp. 49–62, 1999.

[93] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25–53, 2003.

[94] M. Benattou, J.-M. Bruel, and N. Hameurlain, "Generating test data from OCL specification," in *ECOOP Workshops*, 2002.

[95] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Synthetic data generation for statistical testing," in *ASE'17*, 2017, pp. 872–882.

[96] S. Ali, M. Zohaib Iqbal, A. Arcuri, and L. Briand, "Generating test data from OCL constraints with search techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376–1402, 2013.

[97] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is more: A minimalistic approach to UML model-based conformance test generation," in *ICST'08*, 2008, pp. 82–91.

[98] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "Test case generation from natural language requirements based on SCR specifications," in *SAC'13*, 2013, pp. 1217–1222.

[99] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *ICSE'12*, 2012, pp. 815–825.

[100] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *ESEC/FSE'11*, 2011, pp. 416–419.

[101] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*, 2008, pp. 209–224.

[102] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using uppaal-tron: An industrial case study," in *EMSOFT'05*, 2005, pp. 299–306.

[103] D. Varró, "Model transformation by example," in *MODELS'06*, 2006, pp. 410–424.

[104] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. B. Omar, "Search-based model transformation by example," *Software & Systems Modeling*, vol. 11, no. 2, pp. 209–226, 2012.

[105] Z. Balogh and D. Varró, "Model transformation by example using inductive logic programming," *Software & Systems Modeling*, vol. 8, no. 3, pp. 347–364, 2009.

[106] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Model transformation by-example: a survey of the first wave," in *Conceptual Modelling and Its Theoretical Foundations*. Springer, 2012, pp. 197–215.

[107] F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the crowd solve the oracle problem?" in *ICST'13*, 2013, pp. 342–351.

[108] F. Pastore and L. Mariani, "ZoomIn: Discovering failures by detecting wrong assertions," in *ICSE'15*, 2015, pp. 66–76.

[109] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan, "GATE: A framework and graphical development environment for robust NLP tools and applications," in *ACL'02*, 2002, pp. 168–175.

[110] H. F. Ledgard, "A human engineered variant of BNF," *ACM SIGPLAN Notices*, vol. 15, no. 10, pp. 57–62, 1980.

[111] Authors of this paper, "UMTG toolset with source code and replicability package," https://sntsvv.github.io/UMTG/, visited in 2019.

[112] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[113] S. Roy, "Reasoning about quantities in natural language," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2017.

[114] University of Colorado, "Unified Verb Index," 2017. [Online]. Available: http://verbs.colorado.edu/verb-index

[115] "OCLGen Web site." https://OCLGen.github.io.

[116] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," in *ISSTA'98*, 1998, pp. 11–20.

[117] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A challenging model transformation," in *MODELS'07*, 2007, pp. 436–450.

[118] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[119] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding minimal unsatisfiable cores of declarative specifications," in *FM'08*, 2008, pp. 326–341.

[120] Oracle corporation, "Java regular expressions," https://docs.oracle.com/javase/10/docs/api/java/util/regex/Pattern.html, visited in 2019.

[121] "IBM Doors," visited in 2019. [Online]. Available: http://www-03.ibm.com/software/products/en/ratidoor

[122] "Eclipse IDE," https://www.eclipse.org/, visited in 2019.

[123] "Eclipse OCL," http://www.eclipse.org/modeling/, visited in 2019.

[124] "IBM Rhapsody," http://www-03.ibm.com/software/products/en/ratirhapfami, visited in 2019.

[125] "Papyrus Modeling Environment," https://www.eclipse.org/papyrus/.

[126] C. Dania and M. Clavel, "OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints," in *MODELS'16*, 2016, pp. 65–75.

[127] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider, "Potassco: The potsdam answer set solving collection," *AI Communications*, vol. 24, no. 2, pp. 107–124, 2011.

[128] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *ICSE'15*, 2015, pp. 609–619.

[129] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Practical model-driven data generation for system testing," https://arxiv.org/abs/1902.00397, 2019.

[130] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, 1st ed. USA: Cambridge University Press, 2009.

**Chunhui Wang** is Software Engineer in Tools and Infrastructure at Google. He obtained his PhD in Computer Science in 2017 from the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg.

His background concerns automated software testing, including requirements-based testing, testing of embedded, safety-critical systems, and testing of video ads systems.

**Fabrizio Pastore** is Chief Scientist II at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. He obtained his PhD in Computer Science in 2010 from the University of Milano - Bicocca.

His research interests concern automated software testing, including security testing and AI systems testing, based on the integrated analysis of different types of artefacts (e.g., requirements, models, source code, and execution traces). He is active on EU-funded research projects and several industry partnerships.

**Arda Goknil** obtained his Ph.D. in Computer Science at the University of Twente in the Netherlands in 2011. Between 2011 and 2013 he was a postdoctoral fellow at AOSTE research team of INRIA in France. His work at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) of the University of Luxembourg concerns Model-Driven Development, Requirements Engineering, Software Security, Product Line Engineering, and Software Testing.

**Lionel C. Briand** is professor of software engineering and has shared appointments between (1) School of Electrical Engineering and Computer Science, University of Ottawa, Canada and (2) The SnT centre for Security, Reliability, and Trust, University of Luxembourg. He is the head of the SVV department at the SnT Centre and a Canada Research Chair in Intelligent Software Dependability and Compliance (Tier 1).

He holds an ERC Advanced Grant, the most prestigious European individual research award, and has conducted applied research in collaboration with industry for more than 25 years, including projects in the automotive, aerospace, manufacturing, financial, and energy domains. In 2010, he was elevated to the grade of IEEE fellow for his work on testing of object-oriented systems. He was also granted the IEEE Computer Society Harlan Mills award (2012) and the IEEE Reliability Society Engineer-of-the-year award (2013) for his work on model-based verification and testing. His research interests include: Model-driven development, testing and verification, search-based software engineering, requirements engineering, and empirical software engineering.

# APPENDIX A
# COMPLEXITY ANALYSIS OF TEST SCENARIOS GENERATION

In this appendix, we discuss the worst-case time complexity of *GenerateScenariosAndInputs*, the UMTG function that generates test scenarios. We do not discuss the time complexity of procedures implemented to generate object diagrams (i.e., test inputs) because UMTG delegates it to the Alloy analyzer.

The time complexity of *GenerateScenariosAndInputs* depends on the depth-first traversal implemented by *GenerateScenarios*, which is invoked multiple times till the coverage criterion is satisfied or a max number of iterations is reached. Also, it depends on the time complexity of function *maximizeSubTypeCoverage*, which is executed to generate scenarios that satisfy the subtype coverage criterion. *GenerateScenarios*, in turn, invokes six functions (i.e., *coverageSatisfied*, *coverageImproved*, *unsatisfiable*, *addToScenario*, *composeConstraints*, and *negateConstraint*).

In the rest of the appendix, we discuss the time complexity of *GenerateScenarios*, *maximizeSubTypeCoverage*, and the functions invoked by *GenerateScenarios*. Finally, we discuss the time complexity of *GenerateScenariosAndInputs*.

## A.1 Time Complexity of *GenerateScenarios*

*GenerateScenarios* implements a depth-first recursive traversal of a UCTM in which the same node can be visited at most $T+1$ times. Nodes are visited multiple times in the presence of conditional nodes with backward flows (i.e., sequences of nodes departing from the conditional node and reaching a node already visited during the traversal). Backward flows cause loops in the UCTM.

The execution time of *GenerateScenarios* depends on (1) the number of recursive iterations performed, (2) the execution time of the functions being invoked at every recursive iteration and (3) the number of operations (comparisons and assignments) performed within a single iteration of *GenerateScenarios*.

Since *GenerateScenarios* does not contain loops, the number of operations performed within a single iteration of *GenerateScenarios* can be considered constant and thus ignored for the worst-case time complexity analysis.

The time complexity of *GenerateScenarios* can thus be computed as

$$C_{GS} = \mathcal{O}\Big(N_v * \big(1 + (C_{CS} + C_{CI} + C_{US} + C_{AS} + C_{CC} + C_{NC})\big)\Big) \quad (1)$$

with $N_v$ being the number of recursive invocations of *GenerateScenarios* during the traversal of a UCTM, and the terms in the innermost parenthesis (i.e., $C_{CS}$, $C_{CI}$, $C_{US}$, $C_{AS}$, $C_{NC}$, $C_{NC}$) being the time complexity of the functions invoked by *GenerateScenarios* (i.e., *coverageSatisfied*, *coverageImproved*, *unsatisfiable*, *addToScenario*, *composeConstraints*, and *negateConstraint*, respectively). In this section, to avoid confusion with parameter $T$ of *GenerateScenariosAndInputs*, we refer to the time complexity as $C$ instead of $T(n)$ [130].

In this subsection, we focus on the computation of $N_v$. The time complexity of the functions invoked by *GenerateScenarios* is discussed in the following subsections.

Since every recursive call of *GenerateScenarios* traverses only one node of the UCTM, for $T = 0$, *GenerateScenarios* visits every node once, thus having the time complexity as follows

$$C_{GS} = \mathcal{O}\Big(N * \big(1 + (C_{CS} + C_{CI} + C_{US} + C_{AS} + C_{CC} + C_{NC})\big)\Big) \quad (2)$$

with $N$ being the number of nodes in the UCTM.

For $T > 0$, *GenerateScenarios* traverses every node at most $T + 1$ times. To discuss the time complexity for $T > 0$, we first identify the properties of the UCTM that lead to the highest number of recursive iterations of *GenerateScenarios*; we call it the worst-case UCTM.

For $T > 0$, the worst-case UCTM is a UCTM including only condition nodes having one forward and one backward edge, except one condition node having one backward edge and no forward edge. Also, it should have forward edges connecting condition nodes in a sequence, which leads to the longest possible scenario in the UCTM. Finally, all the backward edges should point to the root of the UCTM; this way, each backward edge leads to a traversal of the root, i.e., another traversal of all the nodes of the UCTM. Any other UCTM layout would lead to a traversal of a subset of the UCTM. An example worst-case UCTM is shown in Fig. 19(a).

In the worst-case UCTM, the number of nodes with a backward edge (hereafter, $N_B$) is equal to the total number of nodes of the UCTM (i.e., $N$), while the number of leaf nodes (hereafter, $N_L$) is equal to one. In Section A.5, we discuss whether this configuration leads to the worst-case time complexity for *GenerateScenarios*.

Leaf nodes are always Exit or Abort nodes in the UCTMs generated by UMTG. However, to simplify the discussion, we treat condition nodes with a null forward edge as leafs in Fig. 19(a) and in the rest of this appendix. Basically, we assume that UMTG generates a new scenario when it encounters a null branch. This choice simplifies the presentation without reducing the time complexity; indeed, the absence of leaf nodes augments the portion of condition nodes with backward edges in a UCTM, and, consequently, the time complexity.

Since *GenerateScenarios* traverses all the nodes of the UCTM each time the root node is visited, the value of $N_v$ can be computed as the number of visited root nodes multiplied by the number of nodes that are visited without traversing loops. Fig. 19(b) shows, for different values of $T$, the nodes visited in the worst-case UCTM including 5 nodes. Every node leads to the visit of the root node of the UCTM, and this visit is repeated up to $T$ times. In Fig. 19(b), for T=0, *GenerateScenarios* visits 5 nodes. For $T = 1$, it visits 30 nodes (i.e., the 5 nodes traversed for T=0 and additional 25 nodes reached from any of the nodes in the UCTM). For $T = 2$, it visits 155 nodes (i.e., 125 plus 30). The total number of nodes visited by *GenerateScenarios* is thus captured by the geometric progression $N * \sum_{i=0}^{T} N_c^i$, where $N_c^i$ represents the number of (additional) root nodes visited for an increased value of $T$. Because of the geometric progression,

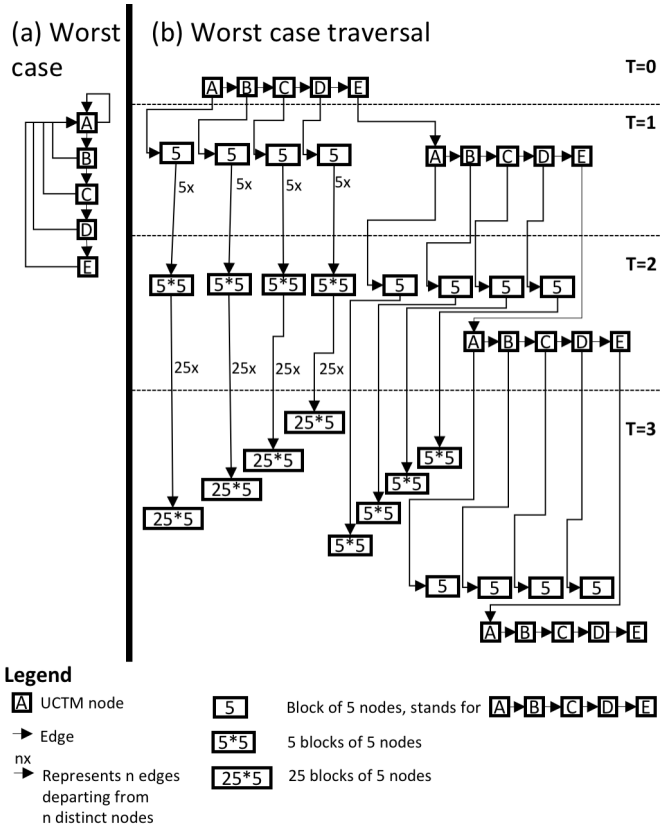


Fig. 19. (a) An Example worst-case UCTM. (b) Traversal of the UCTM, showing the number of nodes visited for different values of T.

the total number of nodes visited by *GenerateScenarios* can be computed as

$$N_v = \frac{N * (1 - N_B^T)}{1 - N_B} \tag{3}$$

However, with $N_B > 0$, the following condition holds

$$\frac{1 - N_B^T}{1 - N_B} < {N_B}^T \tag{4}$$

We can thus further simplify the computation of $\mathcal{O}(N_v)$ as follows

$$\mathcal{O}(N_v) = \mathcal{O}(N * {N_B}^T) \tag{5}$$

## A.2 Time Complexity of *coverageSatisfied*

Function $coverageSatisfied(tm, Prev, Curr)$ checks if a predefined set of coverage targets derived from $tm$ (i.e., the branches of the UCTM and the def-use pairs of the UCTM) is covered by the scenarios that belong to the union of the sets $Prev$ and $Curr$. In the following, we separately discuss the time complexity of the procedures implemented to identify coverage targets (hereafter, $C_{GT}$) and the time complexity of the procedures implemented to determine targets coverage (hereafter, $C_{DT}$).

In Formula 1, the term $C_{CS}$ represents the time complexity of function $coverageSatisfied$; after replacing $C_{CS}$ with the sum of $C_{GT}$ and $C_{DT}$ as indicated above, $C_{GS}$ can be computed as follows

$$C_{GS} = \mathcal{O}\Big(N_v * (1 + (C_{GT} + C_{DT} + C_{CI} + C_{US} + C_{AS} + C_{CC} + C_{NC}))\Big) \tag{6}$$

### A.2.1 Determining $C_{GT}$

UMTG derives coverage targets only on the first invocation of $coverageSatisfied$. Therefore, the time complexity of the procedure for generating coverage targets should not be taken into account for every single recursive iteration of *GenerateScenarios*. This leads to the following formula for $C_{GS}$

$$C_{GS} = \mathcal{O}\Big(C_{GT} + N_v * (1 + (C_{DT} + C_{CI} + C_{US} + C_{AS} + C_{CC} + C_{NC}))\Big) \tag{7}$$

For the branch coverage criterion, the identification of coverage targets requires that the UCTM be traversed without traversing loops (i.e., backward edges), and thus its time complexity is $\mathcal{O}(N)$.

For the def-use coverage criterion, we need to identify all the possible def-use pairs, which, in a UCTM, is lower or equal to the number of unique pairs of nodes. For the def-use coverage, the number of coverage targets $N_{CT}$ can thus be computed as follows

$$N_{CT} = \frac{N * (N + 1)}{2} \tag{8}$$

Consequently, $C_{GT}$ can be computed as follows

$$C_{GT} = \mathcal{O}\left(\frac{N * (N + 1)}{2}\right) \approx \mathcal{O}(N^2) \tag{9}$$

### A.2.2 Determining $C_{DT}$

UMTG implements an optimized version of $coverageSatisfied$ that incrementally determines if the set of coverage targets are covered by the union of $Prev$ and $Curr$. This is done by verifying, at every iteration of *GenerateScenarios*, only the items added to $Curr$.

To check if a coverage target is covered in a scenario, it is necessary to visit, at most, every node in the scenario (e.g, to determine if the scenario contains a specific def-use pair). By definition, the sets $Prev$ and $Curr$ can contain, at most, all the scenarios that can be derived by *GenerateScenarios*. We can thus compute $C_{DT}$ as

$$C_{DT} = \mathcal{O}(N_{CT} * N_{SC}) \tag{10}$$

with $N_{SC}$ being the number of nodes in all the scenarios derived by *GenerateScenarios*.

To simplify the computation of $C_{DT}$, we derive an upper bound for $N_{SC}$. To discuss the computation of the upper bound for $N_{SC}$, we rely on the example in Fig. 20. The example includes the scenarios generated while traversing, for values of $T \leq 3$, a UCTM that has three backward edges (departing from nodes C, D, and E) and three leaves (nodes A, B, and E).

For increasing values of $T$, *GenerateScenarios* traverses additional scenarios originated from the conditional steps with backward edges. In each of these scenarios, we can identify a prefix (hereafter, prefix scenario) starting from the first time the root of the UCTM is visited in the scenario until the last time the root node is visited in the scenario. We can also identify a suffix (hereafter, suffix scenario) starting from the last time the root node is visited for the scenario until

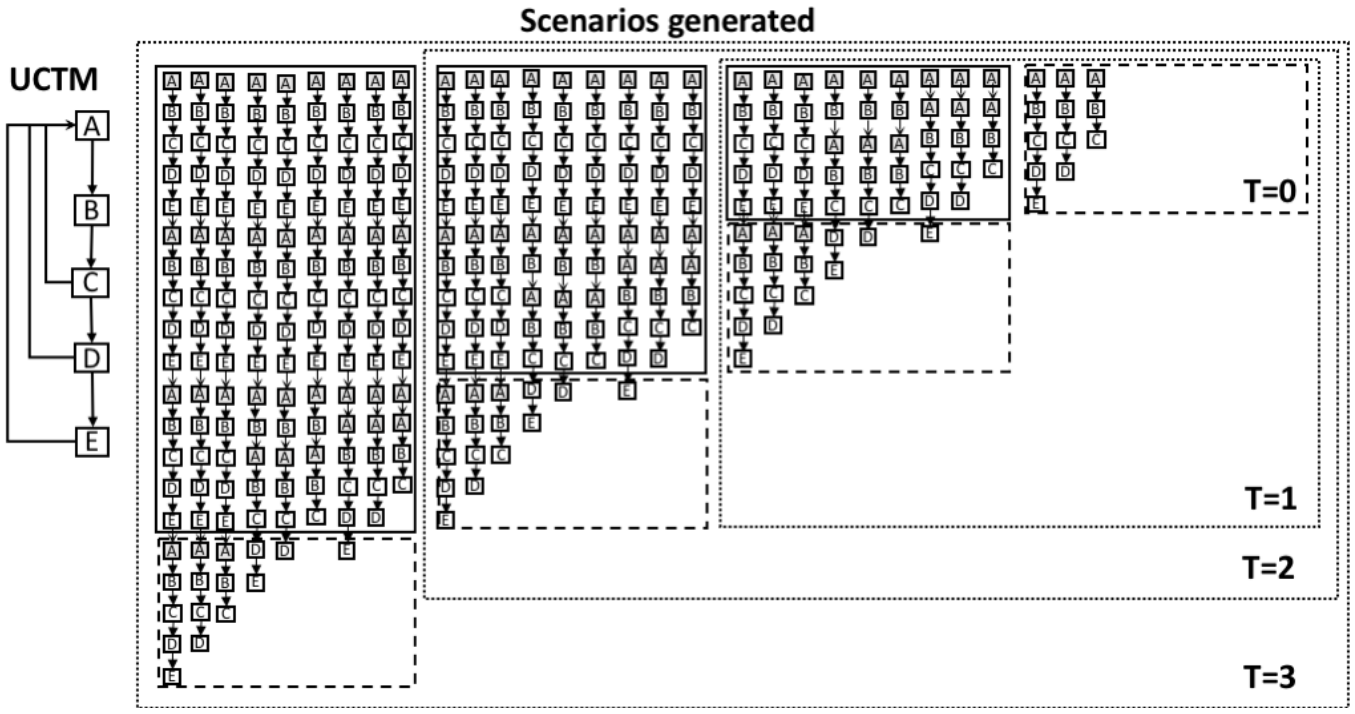


Fig. 20. Example with an upper bound for the number of nodes in the scenarios generated by *GenerateScenarios*, for different configurations of $T$.

a leaf is visited. Each prefix scenario is shared by a number of scenarios that is equal to the number of leaves (i.e., $N_L$) in the UCTM. However, all the $N_L$ scenarios that have the same prefix scenario differ for their suffix scenario.

Please note that worst-case UCTMs contain only condition steps. Therefore, according to the formula ($N_L = N - N_B + 1$), the number of leaves $N_L$ is given by the number of condition steps without a backward edge plus one (the last condition is always a leaf).

In Fig. 20, boxes with black borders show the additional scenarios traversed for increasing values of $T$ (hereafter, $T_i$). The width of these boxes captures the number of the additional scenarios (i.e., $N_B * N_L$), while the height captures the length of the longest prefix scenario (i.e., $N * T_i$). The area of these boxes is thus an upper bound for the number of nodes in the prefix scenarios. It can be computed as

$$\begin{aligned} Area_{prefix} &= \sum_{T_i=1}^{T} N_B * N_L * N * T_i \\ &= N_B * N_L * N * \sum_{i=1}^{T_i} T_i \\ &= N * N_B * N_L * \frac{T * (T+1)}{2} \end{aligned} \tag{11}$$

Prefix scenarios are followed by suffix scenarios. Since the traversal of a root node leads to the generation of a number of scenarios that is equal to the number of leaves, the total number of suffix scenarios is equal to the number of additional root nodes traversed for the increasing values of $T$ multiplied by the number of leaves in the UCTM (i.e., $N_B * N_L$). The total number of such suffix scenarios is thus $N_B * N_L * (T+1)$. In Fig. 20, the maximum number of nodes in the suffix scenarios is depicted by the area of the dashed boxes and can be calculated as

$$Area_{suffix} = N * N_B * N_L * (T+1) \tag{12}$$

The value of $N_{SC}$ can thus be computed as

$$\begin{aligned} N_{SC} &= Area_{suffix} + Area_{prefix} \\ &= N * N_B * N_L * (T+1) + N_B * N_L * N * \frac{T * (T+1)}{2} \\ &= (N * N_B * N_L * T) + (N * N_B * N_L) \\ &+ N_B * N_L * N * \frac{T * (T+1)}{2} \\ &= N * N_B * N_L + N * N_B * N_L * \left( T + \frac{T * (T+1)}{2} \right) \\ &= N * N_B * N_L + N * N_B * N_L * \left( \frac{2T + T^2 + T}{2} \right) \\ &= N * N_B * N_L + N * N_B * N_L * \left( \frac{T^2 + 3T}{2} \right) \\ &= N * N_B * N_L * \left( 1 + \frac{T^2 + 3T}{2} \right) \\ &= N * N_B * N_L * \frac{T^2 + 3T + 2}{2} \end{aligned} \tag{13}$$

The value of $C_{DT}$ can thus be computed as

$$\begin{aligned} C_{DT} &= \mathcal{O}(N_{CT} * N_{SC}) \\ &= \mathcal{O}\left(\frac{N*(N+1)}{2} * N * N_B * N_L * \frac{T^2+3T+2}{2}\right) \\ &= \mathcal{O}\left(\frac{(N^2+N)*N*N_B*N_L*(T^2+3T+2)}{4}\right) \\ &= \mathcal{O}\left(\frac{(N^3*N_B*N_L*(T^2+3T+2))}{4}\right. \\ &\left. + \frac{(N^2*N_B*N_L*(T^2+3T+2))}{4}\right) \\ &\approx \mathcal{O}\left(\frac{(N^3*N_B*N_L*(T^2+3T+2))}{4}\right) \end{aligned} \tag{14}$$

We use the symbol $\approx$ to indicate a simplification enabled by the big-O notation.

We discuss the time complexity for $C_{DT}$ considering $N_B = \frac{N}{2}$, which leads to the higher value for $N * N_B * N_L$, and $N_B = N$, which is the worst case for $N_v$.

With $N_B = \frac{N}{2}$, $C_{DT}$ can be simplified as follows

$$\begin{aligned} C_{DT} &= \mathcal{O}\left(\frac{N^3*N_B*(N-N_B+1)*(T^2+3T+2)}{4}\right) \\ &= \mathcal{O}\left(\frac{N^3*\frac{N}{2}*(\frac{N}{2}+1)*(T^2+3T+2)}{4}\right) \\ &= \mathcal{O}\left(\frac{(\frac{1}{4}N^2+\frac{1}{2}N)*N^3*(T^2+3T+2)}{4}\right) \\ &= \mathcal{O}\left(\frac{\left(N^3*\frac{1}{4}N^2*(T^2+3T+2)\right)}{}\right. \\ &\left.\frac{+\left(N^3*\frac{N}{2}*(T^2+3T+2)\right)}{4}\right) \\ &= \mathcal{O}\left(\frac{\left(\frac{1}{4}N^5*(T^2+3T+2)\right)}{}\right. \\ &\left.\frac{+\left(\frac{1}{2}N^4*(T^2+3T+2)\right)}{4}\right) \\ &= \mathcal{O}\left(\frac{\left(N^5+2N^4\right)*\left(T^2+3T+2\right)}{16}\right) \\ &\approx \mathcal{O}\left(\frac{\left(N^5*\left(T^2+3T+2\right)\right)}{16}\right) \\ &\approx \mathcal{O}\left(N^5\right) \end{aligned} \tag{15}$$

Instead, with $N_B = N$, $C_{DT}$ can be computed as follows

$$\begin{aligned} C_{DT} &= \mathcal{O}\left(\frac{N^3*N_B*N_L*\left(T^2+3T+2\right)}{4}\right) \\ &= \mathcal{O}\left(\frac{N^3*N*1*\left(T^2+3T+2\right)}{4}\right) \\ &= \mathcal{O}\left(\frac{N^4*\left(T^2+3T+2\right)}{4}\right) \\ &\approx \mathcal{O}\left(N^4\right) \end{aligned} \tag{16}$$

Since Formula 15 leads to a higher time complexity than Formula 16, we can conclude that the worst-case UCTM for $C_{DT}$ is charachterized by $N_B = \frac{N}{2}$.

Finally, since function *coverageSatisfied* processes each scenario in $Prev$ and $Curr$ only once, the time complexity $C_{DT}$ should not be taken into account for every recursive iteration of *GenerateScenarios*:

$$\begin{aligned} C_{GS} = \mathcal{O}\Big(&C_{GT} + C_{DT} + N_v * \big(1 \\ &+ (C_{CI} + C_{US} + C_{AS} + C_{CC} + C_{NC})\big)\Big) \end{aligned} \tag{17}$$

### A.3 Time complexity for *coverageImproved*

When a leaf node is reached, *coverageImproved* is invoked by *GenerateScenarios* to check if the new scenario improves coverage. To this end, the nodes of the scenario are compared with all the coverage targets. By construction, *GenerateScenarios* never generates the same scenario twice. Therefore, through all the recursive iterations of *GenerateScenarios*, *CoverageImproved* compares the coverage targets with all the nodes in the scenarios identified by *GenerateScenarios*. The value of $C_{CI}$ can thus be computed as follows

$$C_{CI} = \mathcal{O}(N_{CT} * N_{SC}) \tag{18}$$

Based on Formula 10, the time complexity of *coverageImproved* (i.e., $C_{CI}$) is thus equal to $C_{DT}$

$$C_{CI} = C_{DT} \tag{19}$$

Since *GenerateScenarios* never generates the same scenario twice, we do not need to account for the time complexity of *coverageImproved* for every single recursive iteration of *GenerateScenarios* but for its whole execution against a UCTM. Consequently, we can compute $C_{GS}$ as follows

$$\begin{aligned} C_{GS} = \mathcal{O}\Big(&C_{GT} + C_{DT} + C_{CI} + N_v * \big(1 \\ &+ (C_{US} + C_{AS} + C_{CC} + C_{NC})\big)\Big) \end{aligned} \tag{20}$$

### A.4 Time complexity for *unsatisfiable*

Function $unsatisfiable(pc)$ checks if a path condition (i.e., $pc$) is in the set of path conditions that are known to be unsatisfiable based on the previous executions of the solver. It compares the path condition $pc$ with every path condition in the set. By definition, the set of unsatisfiable path conditions includes, at most, all the path conditions identified in the UCTM by UMTG, i.e., all the path conditions of all the identified scenarios. The worst-case (i.e., highest) number of path conditions identified by UMTG is thus $N_{SC}$. Since we assign an identifier to each path condition (i.e, their hashcode), we can assume that comparing two path conditions has unary cost. The worst-case time complexity of *unsatisfiable* (i.e., $C_{US}$) can thus be computed as follows

$$C_{US} = \mathcal{O}(N_{SC}) \tag{21}$$

#### *A.4.1 Time complexity for* addToScenario, negateConstraint, *and* composeConstraint

Functions *addToScenario*, *negateConstraint*, and *composeConstraint* perform simple operations. Therefore, we ignore them for the computation of the time complexity.

### A.5 Worst cases for *generateScenarios*

Based on the definitions above, the time complexity of *GenerateScenarios* can be computed as follows

$$\begin{aligned} C_{GS} &= C_{GT} + C_{DT} + C_{CI} + \mathcal{O}\Big(N_v * \big(1 + C_{US}\big)\Big) \\ &= C_{GT} + 2 * C_{DT} + \mathcal{O}\Big(N_v * \big(1 + C_{US}\big)\Big) \end{aligned} \tag{22}$$

The last term of the equation above can be simplified as follows

$$\begin{aligned}\mathcal{O}(N_v * (1 + C_{US})) &= \left(N * {N_B}^T\right) \\ &* \left(1 + N * N_B * N_L * \frac{T^2 + 3T + 2}{2}\right) \\ &= N * {N_B}^T \\ &+ N * {N_B}^T * N * N_B * N_L \\ &* \frac{T^2 + 3T + 2}{2} \\ &\approx \frac{T^2 + 3T + 2}{2} * N^2 * {N_B}^{T+1} * N_L\end{aligned} \tag{23}$$

The highest value of the term above depends on the values of $T$, $N_B$, and $N_L$. We discuss the default case for UMTG (i.e., $T = 1$) and the general case (i.e., $T > 1$). For $T = 1$, and $N_B = \frac{N}{2}$, we obtain the following

$$N_L = N - \frac{N}{2} + 1 = \frac{N}{2} + 1 \tag{24}$$

$$\begin{aligned}\mathcal{O}(N_v * (1 + C_{US})) &= \frac{T^2 + 3T + 2}{2} * N^2 * {N_B}^{T+1} * N_L \\ &= 3 * N^2 * \left(\frac{N}{2}\right)^{1+1} * \left(\frac{N}{2} + 1\right) \\ &= 3 * N^2 * \frac{N^2}{4} * \left(\frac{N}{2} + 1\right) \\ &= 3 * \frac{N^4}{4} * \frac{N}{2} + 3 * N^2 * \frac{N^2}{4} \\ &= \frac{3}{8} * N^5 + \frac{3}{4} * N^4 \\ &\approx N^5\end{aligned} \tag{25}$$

With $T = 1$, and $N_B = N$, we obtain the following

$$\begin{aligned}\mathcal{O}(N_v * (1 + C_{US})) &= \frac{T^2 + 3T + 2}{2} * N^2 * {N_B}^{T+1} * N_L \\ &= 3 * N^2 * N^{1+1} * 1 \\ &= 3 * N^4 \\ &\approx N^4\end{aligned} \tag{26}$$

Based on Formula 25 and Formula 26, we can conclude that the higher value for $\mathcal{O}(N_v * (1 + C_{US}))$, with $T = 1$, is thus given by $N_B = \frac{N}{2}$.

With $T \geq 1$, and $N_B = \frac{N}{2}$, we obtain the following

$$\begin{aligned}\mathcal{O}(N_v * (1 + C_{US})) &= \frac{T^2 + 3T + 2}{2} * N^2 * {N_B}^{T+1} * N_L \\ &= \frac{T^2 + 3T + 2}{2} * N^2 * \frac{N^{T+1}}{2^{T+1}} * \left(\frac{N}{2} + 1\right) \\ &\approx \frac{T^2 + 3T + 2}{2} * \frac{N^3}{2} * \frac{N^{T+1}}{2^{T+1}} \\ &\approx \frac{T^2 * N^{T+4}}{2^{T+3}} \\ &\approx N^{T+4}\end{aligned} \tag{27}$$

With $T \geq 1$, and $N_B = N$, we obtain the following

$$\begin{aligned}\mathcal{O}(N_v * (1 + C_{US})) &= \frac{T^2 + 3T + 2}{2} * N^2 * {N_B}^{T+1} * N_L \\ &\approx T^2 * N^2 * N^{T+1} * 1 \\ &\approx N^{T+3}\end{aligned} \tag{28}$$

With $T \geq 1$, the highest value for $\mathcal{O}(N_v * (1 + C_{US}))$ is thus given by $(N_B = \frac{N}{2})$.

Since Formula 9 does not refer to $N_B$, while Formulas 15 and 27 show that the worst-case time complexity is given by $\left(N_B = \frac{N}{2}\right)$, we can conclude that the worst-case time complexity for $GenerateScenarios$ is thus given by $\left(N_B = \frac{N}{2}\right)$.

### A.6 Time complexity for *maximizeSubTypeCoverage*

For each condition node in the scenario generated by *GenerateScenarios*, function *maximizeSubTypeCoverage* invokes the Alloy solver $S$ times, where $S$ is the number of subtypes of the domain entity used in the condition node. $S$ may vary for condition nodes. However, we may assume a constant value that is lower than the number of nodes in the UCTM (i.e., $S < N$). In the computation of the worst-case time complexity for *maximizeSubTypeCoverage*, we ignore the time complexity of the Alloy solver.

The total number of nodes in the scenarios generated by *GenerateScenarios* is given by Formula 13. The worst-case time complexity for *maximizeSubTypeCoverage* (hereafter, $C_{MS}$) can thus be computed as follows

$$C_{MS} = \mathcal{O}(S * N_{SC}) \tag{29}$$

For $(N_B = \frac{N}{2})$, Formula 29 leads to

$$\begin{aligned}C_{MS} &= \mathcal{O}\left(S * N * \frac{N}{2} * \left(\frac{N}{2} + 1\right) * \frac{T^2 + 3T + 2}{2}\right) \\ &\approx \mathcal{O}\left(S * T^2 * \frac{N^3}{4}\right) \\ &\approx \mathcal{O}\left(N^3\right)\end{aligned} \tag{30}$$

For $(N_B = N)$, Formula 29 leads to

$$\begin{aligned}C_{MS} &= \mathcal{O}\left(S * N^2 * \frac{T^2 + 3T + 2}{2}\right) \\ &\approx \mathcal{O}\left(T^2 N^2\right)\end{aligned} \tag{31}$$

The worst-case time complexity for *maximizeSubTypeCoverage* is thus given by $\left(N_B = \frac{N}{2}\right)$.

### A.7 Time complexity for *GenerateScenariosAndInputs*

In this subsection, we discuss the time complexity of *GenerateScenariosAndInputs*. *GenerateScenariosAndInputs* invokes *GenerateScenarios* a number of times equal to $MaxIt$. Also, in the case of subtype coverage, *GenerateScenariosAndInputs* invokes function *maximizeSubTypeCoverage*.

The time complexity of *GenerateScenariosAndInputs* can thus be computed as follows

$$C_{GSI} = C_{GT} + MaxIt * \left(2 * C_{DT} + N_v * (1 + C_{US})\right) + C_{MS} \tag{32}$$

In Formula 32, we do not multiply $C_{GT}$ by $MaxIt$ because the number of coverage targets does not change across different invocations of *GenerateScenarios*.

Based on Formulas 9, 15, 25, and 30, the time complexity of *GenerateScenariosAndInputs* can be computed as follows

$$\begin{aligned}C_{GSI} &= \mathcal{O}\Big(N^2 + MaxIt * \left(2 * N^5 + N^{T+4}\right) + N^3\Big) \\ &\approx \mathcal{O}\big(N^{T+4}\big)\end{aligned} \tag{33}$$

Formula 33 shows that, in the worst-case UCTMs the generation of use case scenarios has at least quintic time complexity, for $T \geq 1$.

Despite the high time complexity, we observe that, based on our empirical evaluation, the generation of use case scenarios with UMTG is feasible in practice. This has two reasons which we detail below.

First, both the number of condition steps, which may potentially lead to loops, and the number of leaves are always lower than half of the total number of use case steps. In our evaluation, the number of use case steps in the UCTMs for *BodySense* and *HOD* is 272 and 136, respectively. The number of condition steps leading to backward edges in these UCTMs is 48 and 25. The number of leaves in the UCTMs is 16 for both case studies. In our case studies, the number of condition steps leading to backward edges is, on average, 18% of the total number of steps, while the number of leaves is 8% of the total number of steps.

Based on our case studies, we may assume that $\left(N_B = \frac{1}{4}N\right)$ and $\left(N_L = \frac{1}{12}N\right)$. Because of their low value, the impact of $N_B$ and $N_L$ on the computation time for *GenerateScenariosAndInputs* is minimal. When we ignore $N_B$ and $N_L$, for $T = 1$ and $MaxIt = 10$, we can compute the value of $C_{GSI}$ using Formulas 9, 14, and 23 as follows

$$
\begin{aligned}
C_{GSI} &= C_{GT} + MaxIt * \Big(2 * C_{DT} + N_v * (1 + C_{US})\Big) + C_{MS} \\
&= \mathcal{O}\Bigg(N^2 + 10 * \Bigg(2 * \frac{N^3 * N_B * N_L * (T^2 + 3T + 2)}{4} \\
&= + \frac{(T^2 + 3T + 2) * N^2 * {N_B}^{T+1} * N_L}{2}\Bigg) + N^3\Bigg) \\
&\approx \mathcal{O}\Bigg(N^2 + 10 * \Bigg(2 * \frac{(N^3 * 6)}{2} + \frac{6}{2} * N^2\Bigg) + N^3\Bigg) \\
&= \mathcal{O}\Bigg(N^2 + 10 * \Bigg(\big(6 * N^3\big) + 3 * N^2\Bigg) + N^3\Bigg) \\
&\approx \mathcal{O}\Bigg(60 * N^3\Bigg)
\end{aligned}
\tag{34}
$$

Finally, the total number of nodes in the UCTMs for testing the real systems in our case studies is not high. This is mostly due to the fact that the use case specifications are manually written and describe the high-level behavior of the systems. For example, the largest UCTM in our case studies, which has been generated for *BodySense*, includes 154 nodes. With a relatively small number of nodes, UCTMs can be processed fast by using modern computing systems (e.g., laptops), even in the presence of $\mathcal{O}(N^3)$ complexity.