Automated Generation of Constraints from Use Case Specifications to Support System Testing

Chunhui Wang, Fabrizio Pastore, Lionel Briand
SnT Centre for Security Reliability and Trust - University of Luxembourg
{chunhui.wang,fabrizio.pastore,lionel.briand}@uni.lu

***Abstract*—System testing plays a crucial role in safety-critical domains, e.g., automotive, where system test cases are used to demonstrate the compliance of software with its functional and safety requirements. Unfortunately, since requirements are typically written in natural language, significant engineering effort is required to derive test cases from requirements.**

In such a context, automated support for generating system test cases from requirements specifications written in natural language would be highly beneficial. Unfortunately, existing approaches have limited applicability. For example, some of them require that software engineers provide formal specifications that capture some of the software behavior described using natural language. The effort needed to define such specifications is usually a significant deterrent for software developers.

This paper proposes an approach, *OCLgen*, which largely automates the generation of the additional formal specifications required by an existing test generation approach named *UMTG*. More specifically, *OCLgen* relies on semantic analysis techniques to automatically derive the pre- and post-conditions of the activities described in use case specifications. The generated conditions are used by *UMTG* to identify the test inputs that cover all the use case scenarios described in use case specifications. In practice, the proposed approach enables the automated generation of test cases from use case specifications while avoiding most of the additional modeling effort required by *UMTG*.

Results from an industrial case study show that the approach can automatically and correctly generate more than 75% of the pre- and post-conditions characterizing the activities described in use case specifications.

I. Introduction

System testing plays a crucial role in the development process of embedded software in safety critical domains, e.g., automotive, because system test cases are used to demonstrate that the software meets its functional and safety requirements. This practice is enforced by safety standards, e.g. ISO 26262 [1].

Software requirements are commonly expressed by means of natural language and, for this reason, system test cases are typically derived manually by the software engineers. Unfortunately, manual generation of system test cases is an expensive and error-prone activity.

The adoption of automated techniques that reduce testing costs and provide guarantees about requirements coverage could be an important advantage for companies developing safety-critical software. However, most of the existing approaches that generate system test cases from requirements rely on simplistic natural language processing solutions that limit their applicability.

Some approaches are based on the detection of specific keywords (e.g., *if* and *then*) that enable the identification of different use case scenarios [2], [3], [4]. The identified scenarios are used to derive test cases that correspond to sequences of use case steps. These test cases are, however, abstract and only provide high-level guidance to the testers. The mere identification of keywords, in fact, does not enable the determination of actual input values to be used during testing. For example, these approaches cannot automatically determine that it is necessary to input a tension of 11 Volts in order to check if the system complies with the specification *if the voltage is below 12 then signal low tension*.

Other approaches [5], [6] require that software specifications be written according to a controlled natural language (CNL) [7]. These approaches implement textual transformation rules specific to the CNL considered that translate CNL specifications into formal specifications. The generated specifications are then processed to generate test inputs automatically (e.g., using constraint solving). The CNL language supported by these techniques is typically very limited as it only enables engineers to use few verbs in requirements specifications.

Finally, other approaches that generate executable test cases do not require a restricted vocabulary, but entail additional modeling effort [8], [9]. These approaches rely on the identification of keywords to determine the sequences of use case steps to consider during testing, but, in addition, they expect that software engineers provide complementary formal specifications that enable the automatic generation of concrete test inputs. For example, UMTG (Use Case Modelling for System Tests Generation) requires that engineers provide constraints written using the Object Constraints Language (OCL) [10] to capture the meaning of conditional sentences [8].

In this paper we propose *OCLgen*, a technique that automatically derives constraints that capture either the effects (post-conditions) of the activities described in a step of a use case specification, or the pre-conditions described by the conditional statement of a use case specification. We focus on use case specifications because we build on *UMTG* and they are frequently adopted to capture software requirements [11].

OCLgen complements the test generation approach named *UMTG* by automating the definition of constraints expressed using the OCL language, an activity that in *UMTG* is manually performed by software engineers. *OCLgen* thus contributes to further automating the generation of system test cases from artifacts produced during requirements analysis (i.e., use case

specifications and the domain model of the system).

A core contribution of the paper is the definition of text transformation rules that rely on automated semantic analysis techniques to generate OCL constraints without requiring the adoption of a limited, constrained natural language.

OCLgen exploits recent advances in semantic analysis research, such as semantic role labeling (SRL) techniques [12] and lexicons [13], [14]. SRL techniques automatically determine the role of the nouns appearing in sentences (e.g., the actor most directly affected by the action described in the sentence). Lexicons enable the identification of synonyms.

OCLgen generates OCL constraints using text transformation rules based on the roles provided by SRL techniques. More precisely, *OCLgen* uses the roles identified by SRL to select the variables and operators to be included in the generated OCL constraints. For instance, the noun that is indicated by SRL as the actor most directly affected by the action described in the sentence typically corresponds to the object whose state should be captured by a post-condition.

OCLgen also relies on the VerbNet [14] and WordNet [13] lexicon to determine if two verbs have the same meaning and, as a consequence, if a same text transformation rule can be adopted to process both. This choice enables the processing of a big portion of the English vocabulary with a limited set of transformation rules. The same rule can be reused to process multiple verbs having the same meaning.

An industrial case study suggests that our approach is promising as we can automatically and correctly generate 75% of the constraints, the remaining 25% cannot be generated due to a lack of precision in the use case specifications.

The paper proceeds as follows. Section II provides background information about UMTG, semantic role labelling, and synonyms detection. Section III presents an overview of the approach. Sections IV to IX give additional details about the algorithmic choices behind *OCLgen*. Section X discusses the completeness of the approach with respect to the English vocabulary. Section XI provides an overview of the empirical results obtained. Section XII discusses related work. Section XIII concludes the paper.

II. Background

A. System Testing with UMTG

UMTG is a technique that automatically generates executable test cases by processing the use case specifications and the domain model of a software system [8]. *UMTG* requires that use case specifications are written according to the RUCM format. RUCM is a use case format that provides restriction rules and keywords constraining the use of natural language that ease the automated extraction of data from use case specifications. For details, the reader is referred to [15].

Table I shows a portion of the use case specifications used to illustrate the *UMTG* approach in another work [8]. These specifications describe the behaviour of a car seat occupant classification system named *BodySense*™.

The *basic flow* section in the RUCM template captures the main success scenario of a use case specification, sections named *specific alternative flow* capture the sequences of steps taken when a certain condition does not hold (e.g., the occupant class for airbag control not being valid in the case of Table I), while *bounded alternative flows* capture activities performed when a certain condition becomes true in a given range of steps (e.g., a voltage error being detected before the basic-flow steps number two or three in the case of Table I).

TABLE I
USE CASE *Identify Initial Occupancy Status of a Seat*

1	**Precondition**
2	The system has been initialized
3	**1.1 Basic Flow**
4	1. The seat SENDS occupancy status TO the system.
5	2. INCLUDE USE CASE Classify occupancy status.
6	3. The system VALIDATES THAT the occupant class for airbag control is valid.
7	4. The system SENDS the occupant class for airbag control TO AirbagControlUnit.
9	Postcondition: The occupant class for airbag control has been sent.
11	**1.2 Bounded Alternative Flow**
12	RFS 2-3
13	1. IF voltage error is detected THEN
14	2. The system resets classification filters.
15	3. RESUME STEP 1.
16	4. ENDIF
17	Postcondition: Classification filters have been reset.
18	**1.3 Specific Alternative Flow**
19	RFS 3
21	2. The system SENDS the previous occupant class for airbag control TO AirbagControlUnit.
22	3. RESUME STEP 1.
23	Postcondition: The previous occupant class has been sent.

In Table I, capital letters are used to highlight the RUCM keywords used to support automated processing in *UMTG*. The keyword *VALIDATES THAT*, for example, indicates that the system checks if a condition holds. The keyword *IF ... THEN*, instead, is used in bounded alternative flows to indicate the condition that activates the alternative flow. The keyword .. *SENDS..TO* in Line 4 indicates that an input has been sent to the system.

To support test case generation, *UMTG* requires that engineers provide constraints written in the Object Constraint Language (OCL) for every conditional clause appearing in the use case specification (i.e., pre-conditions and steps containing the keywords *VALIDATES THAT* and *IF ... THEN*). Since internal steps may affect the evaluations in conditional steps, *UMTG* also requires that engineers provide OCL constraints that capture the post-conditions of every internal step. A portion of the constraints required for the example use case specification is shown in Table II.

To automatically generate test cases, *UMTG* first generates a model capturing the sequences of steps appearing in a use case specification. Figure 2 shows the model generated from the specifications in Table I. *UMTG* then generates a different test case for each path from the start step to an exit step of the model. The test inputs of a test case are then identified by solving the path condition that joins all the OCL conditions associated with the steps appearing in the path. For example, to generate a test case that covers the sequence of use case steps in grey in Figure 2, *UMTG* joins the conditions appearing on lines 2, 13 and 14, and generates an instance of the domain

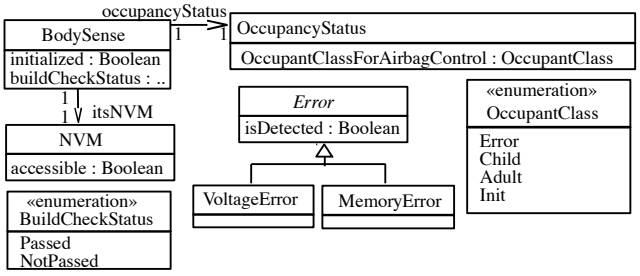

Fig. 1. Partial domain model for $BodySense$

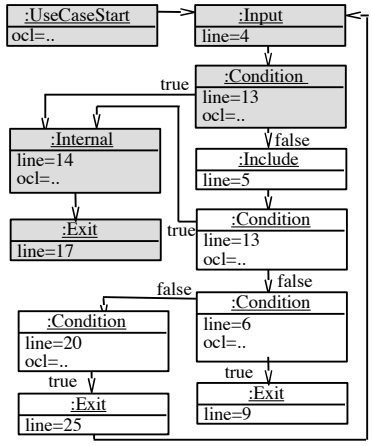

Fig. 2. Testing model generated by *UMTG*

Path constraint:

BodySense.allInstances()->forAll(b | b.initialized = true) **and** VoltageError.allInstances()->forAll(v | v.detected = true) **and** BodySense.allInstances()→forAll(b | b.occupancyStatus.occuPant-ClassForAirbagControl= null)

Object diagram generated by the OCL constraint solver:

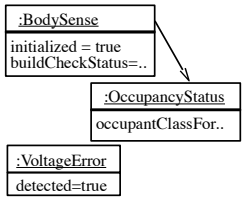

Fig. 3. Object diagram generated by constraint solving.

model that is a solution for the path condition (Figure 3).

UMTG then uses a mapping table to translate the abstract test input generated (i.e., the domain model instance) into an executable test case containing the concrete inputs to send to the system under test. The executable test case generated for the example above initializes the system, triggers a voltage error (e.g., reduces the voltage of the circuit board) and then checks if *BodySense*™ properly reacts (e.g., resets classification filters).

Although *UMTG* can automatically generate test cases from specifications, the effort required to specify additional OCL constraints may prevent engineers from adopting *UMTG*.

B. Natural Language Processing and Semantic Role Labeling

Natural language processing (NLP) techniques extract structured information from documents written in natural language. NLP techniques are implemented as a pipeline that executes different analyses, for example tokenization, morphology analysis, syntax analysis (e.g., part-of-speech tagging) and semantic analysis (e.g., semantic role labeling) [16].

In this paper we rely on semantic analysis, more precisely automated semantic role labelling (SRL) techniques. SRL techniques process a sentence and automatically determine the roles played by the phrases[1] appearing in it, e.g., they can identify the actor who is affected by an activity described in a sentence [16]. Our choice is based on the observation that determining the role of phrases is necessary for deriving pre- and post-conditions. For example, to derive a post-condition for a use case step, it is necessary to know who is affected by the consequences of the activity.

Most of the existing NLP-based test generation [17], [18], [19] and constraint generation [20], [21] approaches rely on syntax analysis techniques like part-of-speech (POS) tagging, which helps identify subjects and verbs, and dependency parsing, which helps identify the determiners associated with a word. The information provided by these techniques is typically used to identify the concepts appearing in software models or code (e.g., class names), but they are not capable of directly identifying the role of phrases appearing in sentences.

If we consider the two sentences *The system starts* and *The system starts the database*, SRL techniques can determine that the actors affected by the actions described in these sentences are *the system* and *the database*, respectively. This information cannot be captured by POS tagging and dependency parsing approaches, because 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.

A few automated SRL tools exist [22], [23], [24]. These tools apply machine learning algorithms in order to perform SRL, and they differ in the models adopted to capture semantic roles. Semafor [22], [25], and Shalmaneser [23] are based on the FrameNet model, while the CogComp NLP pipeline (hereafter CNP [24]) is based on the PropBank [26] and the NomBank models [27], [28]. According to our experience, CNP is the only tool still under active development and for this reason it has been used to implement the technique proposed in this paper. CNP can be invoked either programmatically or by using a Web interface [29].

In the following paragraphs we provide the details of the SRL models adopted in the context of this paper, which are PropBank and NomBank.

SRL tools based on the PropBank corpus [26] (e.g., CNP) tag the words appearing in sentences with abstract keywords (e.g., *A0*, *A1*, *A2*, *AN*) that capture the roles of the words appearing in a sentence. The abstract keyword *A0* always indicates who (or what) is performing an action, while *A1* indicates the participant most directly affected by an action. The semantics of the other roles is specific to each verb, although there are some commonalities. For example, *A2* is often used to indicate the end state of an action.

Tools based on PropBank use the keyword *A1* to tag the term *The counter* in the following three sentences *The system increases the counter*, *The counter was increased by the system*, *The counter increased 5%*. Also, in the two sentences *The system starts* and *The system starts the database*, SRL tools based on PropBank use the keyword *A1* to tag the term *The system* in the first sentence and the term *the database* in the second sentence, thus enabling the automated identification of the component which has been started.

The ProbBank model also includes additional semantic

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

TABLE II
SOME CONSTRAINTS FOR THE USE CASE '*Identify Initial Occupancy Status of a Seat*'.

Line	**Condition in the Use Case**	**Corresponding OCL Constraint**
2	The system has been initialized	$BodySense.allInstances() \rightarrow forAll(b \mid b.initialized = true)$
6	the occupant class for airbag control is valid	$BodySense.allInstances() \rightarrow forAll(b \mid b.occupancyStatus.occupantClassForAirbagControl <> OccupantClass :: Error)$
13	voltage error is detected	$VoltageError.allInstances() \rightarrow forAll(v \mid v.detected = true)$
14	The system resets classification filters	$BodySense.allInstances() \rightarrow forAll(b \mid b.occupancyStatus.occupantClassForAirbagControl = null)$

TABLE III
PROPBANK ADDITIONAL SEMANTIC ROLES USED IN THE PAPER.

Identifier	**Definition**
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-PRD	Secondary predicate with additional information about A1.

roles that are not verb specific. These roles are labeled with general keywords and match adjunct information present in different sentences (e.g., the keyword *AM-NEG* is used to indicate that the verb is negated). Some of the ProbBank additional semantic roles that are used in this paper are shown in Table III.

The NomBank model, instead, captures the roles of nouns, adverbs, and adjectives appearing in noun phrases. NomBank relies on the same keywords adopted by PropBank.

When processing the sentence *The system schedules the daily jobs*, the analysis based on ProbBank enables an SRL tool (e.g., CNP) to indicate that the noun phrase *the daily jobs* plays the role *A1*. The analysis based on NomBank, instead, enables SRL tools to provide complementary information indicating that the term *jobs* is the main noun, while the term *daily* provides temporal information (labelled as *AM-TMP*).

C. Similarity Detection

The PropBank model does not help determine that two distinct sentences describe similar concepts. When processing the sentences *The system stopped the database*, *The system halted the database* and *The system terminated the database* an SRL tool based on PropBank labels *the database* with the tag *A1* thus indicating that the database is the thing that changes its state in the three sentences. However, the tag *A1* does not enable us to determine that the three sentences have similar meanings.

One of the available options to identify verbs with semantic similarities is the VerbNet lexicon [30], which clusters together verbs that have a common semantics. Each verb class is provided with a set of *role patterns* that capture the roles that commonly appear in clauses including these verbs. For example *<A1,V>* and *<A0,V,A1>* are two of the *role patterns* of the VerbNet class *stop-55.4*, which includes, among the others, the verbs *to stop*, *to halt* and *to terminate*. The role pattern *<A1,V>* indicates that a sentence can contain only the verb and the actor whose state is altered (i.e., in the case of the verb *to stop*, the actor that has been stopped). The role pattern *<A0,V, A1>* indicates that a sentence can contain the actor performing the stopping action (i.e, the phrase tagged as A0), the verb, and the actor being stopped (i.e., the phrase tagged as A1). Examples of these two frames are *the database stops* and *the system stops the database*, respectively. In this paper, we rely on VerbNet version 3.2 [31], which includes 272 verb classes and 214 subclasses (a subclass is a subset of the verbs of a class that share specific semantics when used with a particular *role pattern*).

VerbNet uses a role model different than PropBank. However, for each set of roles associated to verbs appearing in VerbNet, there exist a mapping to the corresponding roles in the PropBank model [14]. For the sake of simplification, in this paper we use PropBank role labels only.

All the verbs appearing in a class are guaranteed to share a common set of role patterns, which helps defining text transformation rules based on semantic roles that can be reused across all the verbs of a same class. However, not all the verbs in a class are guaranteed to be synonym (e.g. the VerbNet class *stop-55.4* includes also the verb *repeat*). The state of the art approach to identify verb synonyms is WordNet [32], which is a database of lexical relations that uses sets of words (called synsets) to cluster together words with the same meaning.

III. APPROACH OVERVIEW

The approach proposed in this paper, *OCLgen*, complements *UMTG* [8] by automatically generating the OCL constraints that in *UMTG* are manually written by engineers. *UMTG* requires that engineers use OCL constraints to capture two kinds of information: (1) the effect that the activities described in internal steps have on the state of system (i.e., the postcondition of internal steps) and (2) the pre-conditions described in conditional steps and use case headings. *OCLgen* aims to automate this process.

OCLgen requires the same inputs as *UMTG*: (1) textual use-case specifications written according to RUCM [15], and (2) a domain model of the system as a UML class diagram. *OCLgen* automatically generates an OCL constraint for every use case step that *UMTG* requires to be specified with an OCL constraint.

To generate the OCL constraint that captures the meaning of a use case step *OCLgen* leverages the capabilities of existing SRL and similarity detection techniques. More specifically, *OCLgen* selects the elements that should appear in an OCL constraint based on the roles identified by SRL, decides the comparison operators to be used in the constraints based on the verb identified by SRL, introduces additional operators (e.g., negation) based on the additional semantic roles proposed by SRL.

In *OCLgen*, the generation of OCL constraints is driven by a set of transformation rules, each one tailored to the meaning of a verb (*OCLgen* applies the rule corresponding to the verb appearing in the use case step). To deal with general use case specifications that are not limited to a very limited constrained natural language, while keeping a manageable set of transformation rules, *OCLgen* relies upon similarity detection techniques. More precisely, the same rule is applied to all the verbs belonging to the same VerbNet class that are synonyms.

OCLgen executes four activities for each use-case step to be translated to an OCL constraint:

1) Execute the SRL toolset of the University of Illinois (CNP) on the sentence appearing in the use case step;
2) Based on the verb identified by CNP, select a set of applicable transformation rules;
3) Execute each of the selected transformation rules. Each transformation rule generates a candidate OCL constraint, and assigns it a score.
4) Assign to the use case step the OCL constraint with the highest score.

A sentence may lead to multiple OCL constraints (e.g., we may apply multiple transformations rules to the same use case step) and we want to generate the most precise constraint. For this reason we adopt a scoring mechanism (activity 3 above) which assigns the highest score to the OCL constraint that reuses most of the information available in a use case step, and is detailed in Section IX.

Table IV reports some examples of use case steps that are used in the paper to describe how *OCLgen* works. Table IV indicates the roles identified by CNP and the corresponding OCL constraints that are generated by *OCLgen*. For example in the case of sentence S1, CNP identifies *"The system"* in the role A0 (actor who performs the action), *"sets"* as verb, *"the occupant class for airbag control"* as A1, and *"to Init"* as *A2* (final state). In the case of conditional steps, *OCLgen* ignores the RUCM keyword *The system VALIDATES THAT*, which is not useful to determine the corresponding pre-condition.

The following sections discuss the details of *OCLgen*.

IV. Format of the OCL constraints

The constraints described in the use case specifications of embedded systems are typically simple. For example, pre-conditions and conditional steps usually describe safety checks ensuring that the environment has been properly set up (e.g., Line 2 in Table II) or that values have been properly derived (e.g., Line 6), while internal steps often describe updates of state variables (e.g., Line 14). These types of constraints can be expressed using the OCL language according to the pattern reported in Figure 4, which allows us to capture assignments, equalities, and inequalities. *OCLgen* focus on the generation of constraints expressed with that pattern.

These OCL constraints include a class name, an optional selection part, and an expression that should hold for all the object instances `i` belonging to the selection (e.g., all the instances of the class). The expression contains a left-hand side variable, an OCL operator, and a right-hand side term (either another variable or a literal). The optional selection part simply selects a subset of all the available instances based on subtypes of the class `CLASS`.

CONSTRAINT = **CLASS** .allInstances() **[SELECTION]** .forAll(**EXPR**)
EXPR = i | i.**VARIABLE OPERATOR RHS**
VARIABLE = **ATTR** | **ASSOC** { .**ATTR** | **ASSOC** }
RHS = **VARIABLE** | **LITERAL**;
SELECTION = → select(e | **TYPESEL** { and **TYPESEL** })
TYPESEL = not e.TypeOf(**CLASS**)
Note: This pattern is expressed using a simplified EBNF grammar [33] where non-terminals are bold and terminals are not bold. **CLASS** stands for a class name appearing in the domain model, **LITERAL** is a OCL literal (e.g., '1' or 'a'), **ATTR** is an attribute of a class in the domain model, **ASSOC** is the name of an association end appearing in the domain model, **OPERATOR** is a math operator (-, +, =, $<$, $\leq$, $\geq$, $>$).

Fig. 4. Pattern of OCL constraints generated by *OCLgen*.

Require: srl, a sentence annotated with the different roles identified by SRL
Require: $systemClass$, the main class of the system
Ensure: $< ocl, score >$, an OCL constraint with a score
1: **function** TRANSFORM(srl)
2: $lhsVariables$ ←process srl and identify a set of variables that might appear
3: in the left-hand side of the OCL constraint
4: **for each** LHS *in* $lhsVariables$ **do**
5: RHS ← identify the term to put on the right-hand side
6: OP ←identify the operator to use in the OCL constraint
7: SEL ←if needed, build a sub-expression including the selection operator
8: **if** $RHS \neq null$ *and* $OP \neq null$ **then**
9: ocl ←build the OCL constraint using LHS, SEL, OP and RHS
10: $score$ ←calculate the score of the OCL constraint
11: $ocls \leftarrow ocls \cup < ocl, score >$
12: **end if**
13: **end for**
14: $bestOcl$ ← select the OCL constraint with the best score from the list $ocls$
15: **return** $bestOcl$
16: **end function**

Fig. 5. Template structure shared by all the transformation rules.

V. Transformation rules

OCLgen implements a set of verb-specific transformation rules that enable the automated generation of an OCL constraint from a use case step written in natural language. The transformation rules implemented in *OCLgen* share a common structure and execute a predefined set of activities that are described in this section. Each transformation rule is associated with a set of English verbs and is executed if one of the verbs in the set appears in the use case step to process.

In Section X we discuss to what extent *OCLgen* covers all the verbs of the English language. In the following sections, to describe *OCLgen* we focus on the transformation rules identified for the verbs *to be*, *to set*, and *to enable*.

Figure 5 shows the common algorithmic steps followed by all the transformation rules implemented by *OCLgen*.

All the transformation rules rely on SRL roles to identify the left-hand side variables (hereafter lhs-variables), operators, selection elements, and right-hand side terms (hereafter rhs-terms). For example, the phrases tagged with the role A1 are usually used to determine the variable on the left-hand side of an OCL expression.

All the transformation rules first identify a set of variables that might be used on the left-hand side of the OCL expression (Line 2 in Figure 5). Instead of identifying a single variable for each OCL constraint to be generated, we opted for identifying

TABLE IV
SOME CONSTRAINTS FROM THE *BodySense*™ CASE STUDY, WITH TAGS GENERATED BY SRL.

#	Sentence with SRL tags	Corresponding OCL Constraint
S1	{The system}$_{A0}$ {sets}$_{verb}$ {the occupant class for airbag control}$_{A1}$ {to Init}$_{A2}$	$BodySense.allInstances() \rightarrow forAll(b \mid b.itsOccupancyStatus.occupant ClassForAirbagControl = OccupantClass :: Init)$
S2	{The system VALIDATES THAT}$_{ignored}$ {the NVM}$_{A1}$ {is}$_{verb}$ {accessible}$_{AM-PRD}$	$BodySense.allInstances() \rightarrow forAll(i \mid i.itsNVM.isAccessible = true)$
S3	{The system}$_{A0}$ {sets}$_{verb}$ {temperature errors}$_{A1}$ {to detected}$_{A2}$	$TemperatureError.allInstances() \rightarrow forAll(i \mid i.isDetected = true)$
S4	{The system VALIDATES THAT}$_{ignored}$ {the build check}$_{A1}$ {has been passed}$_{verb}$	$BodySense.allInstances() \rightarrow forAll(i \mid I.buildCheckStatus = BuildCheckStatus :: Passed)$
S5	{The system VALIDATES THAT}$_{ignored}$ {no}$_{NOM-NEG}$ error {{except voltage errors}$_{NOM_{NP}}$ and {memory errors}$_{NOM_{NP}}$}$_{NOM-A2}$}$_{A1}$ {is detected}$_{verb}$	$Errors.allInstances() \rightarrow select(e \mid !e.typeOf(VoltageError) and !e.typeOf(MemoryError)) \rightarrow forAll(i \mid i.isDetected = false)$

TABLE V
SUPPORT ROLES APPEARING IN OUR EXAMPLES

Rule (identified with the corresponding verb)	Support roles
to be	AM-PRD
to enable	AM-MNR
to set	A2
$META - VERB - RULE$	AM-PRD,Verb

a set of potential variables to be used for generating a corresponding set of OCL constraints and then select the OCL constraint with the highest score.

The algorithm iterates over the lhs-variables identified to generate a distinct OCL constraint for each one (Lines 4 to 13 in Figure 5). However, if the operator or the rhs-term of the OCL expression is not identified, then the OCL constraint is not generated (Line 8).

If all the required elements are available, then the algorithm builds the OCL constraint according to the pattern reported in Figure 4 (Line 9 in Figure 5) and finally calculates the associated score (Line 10). Section IX provides details about the formula used to calculate the score.

In addition to transformation rules specific for the different verbs that can appear in use case steps, *OCLgen* includes a general transformation rule that is applied for any verb. We call this *meta-verb transformation rule*. This transformation rule is based on the observation that use case steps are often translated to OCL constraints whose lhs-variable is an attribute whose name matches the name of the verb (or they are similar). Additional details about this transformation rule are provided in the rest of the paper. For every use case step, *OCLgen* applies both the verb-specific transformation rule and the meta-verb transformation rule and then selects the OCL constraint with the highest score.

Sections VI to IX describe the different operations performed during the execution of the algorithm, i.e., identifying the lhs-variables, identifying the rhs-term, identifying the operators to appear in the OCL constraint, and finally scoring the generated constraints.

VI. IDENTIFICATION OF THE VARIABLES TO BE USED ON THE LEFT-HAND SIDE OF THE OCL EXPRESSION

All the transformation rules identify the lhs-variables by performing the same set of operations, which are captured by function *findVariables*, reported in Figure 6. The behavior of this function depends on a set of *support roles*. Support roles correspond to a subset of SRL roles specific to each

```
1:  function FINDVARIABLES(srl, systemClass, SupportRoles)
2:      //Find an attribute of the system class with a name that matches A1
3:      term_A1 ← preprocess(srl.get(A1))
4:      attr ← findAttribute(systemClass, term_A1)
5:      Vars ← Vars ∪ attr
6:      //Find a class with a name that matches A1
7:      class ← findClass(term_A1)
8:      if class ≠ null then
9:          for role in SupportRoles do
10:             //Check if the class contains attributes matching the support roles
11:             attr ← findAttribute(class, srl.get(role))
12:             if attr ≠ null then
13:                 Vars ← Vars ∪ attr
14:             end if
15:         end for
16:     end if
17:     //Find boolean attributes that match the additional roles to process
18:     for var in Vars do
19:         Vars ← Vars ∪ identifyBoolVars(srl, var, SupportRoles)
20:     end for
21: end function
```

Fig. 6. The algorithm to identify lhs-variables.

transformation rule that help the identification of lhs-variables. Table V shows the support roles for the different transformation rules considered in the examples reported in the paper. The use of support roles in *OCLgen* is described in the following paragraphs.

In general, we expect that the phrase tagged as A1 (i.e., the actor that is affected by the action described in the use case step) provides part of the information required to identify the lhs-variable (e.g., it may indicate the class it belongs to), while the support roles provide information to further characterize this variable (e.g., a support role may indicate which attribute of the class identified by A1 should appear in the lhs-variable).

The strategy adopted for identifying lhs-variables is strongly influenced by the practices commonly followed by engineers when producing a domain model [11]. We have identified five modelling choices that affect the resulting domain model and, as a consequence, the format of OCL constraints:

1) In the case of embedded systems, the domain model often includes a system class with attributes that capture information about the system state. This mainly depends on the fact that use case specifications often describe abstract state-based behaviors, and thus state information needs to be included in the domain model. *BodySense* is the system class for the model in Figure 1.
2) In general, the concepts appearing in requirements specifications are modeled as classes, associations or attributes in the domain model.

3) The names of the attributes and associations appearing in the domain model are usually consistent with the phrases appearing in use case specifications.
4) Sometimes additional classes are introduced in the domain model to group concepts that are modelled using attributes.
5) Boolean attributes often have names that are representative of a system state (e.g., *isAccessible*), while in the case of other types of attributes state information is captured by the values assigned to the attribute (see for example *buildCheckStatus* appearing in S4 in Table IV).

To identify attributes while accounting for the five points above, *OCLgen* adopts a flexible approach: it does not try to determine which choices have been made during modeling (e.g., whether attributes appear in a system class or in another class), but identifies lhs-variables assuming that any of those five choices may possibly hold, as described in the following.

To deal with the presence of a system class (Item 1 above), *OCLgen* checks if the system class (the name is provided by software engineers) contains an attribute whose name best matches the phrase tagged as A1 (Line 4 in Figure 6).

To deal with concepts appearing in classes different than the system class (Item 2), *OCLgen* uses the phrase tagged as A1 to identify the class that should contain the lhs-variable (Line 7) and then looks for an attribute name that best matches one of the terms tagged with support roles (Lines 8 to 16).

To identify the class name to use in the OCL constraint, *OCLgen* looks for a class whose name has a minimal distance from the phrase labeled as A1. To this end, *OCLgen* relies on the Needleman-Wunsch string alignment algorithm [34], which maximizes the matching between characters and allows for some degree of discrepancy between the class name and the concept in the use case specifications.

In the case of sentence S3 in Table IV, function *findVariables* first detects that the class to consider is *TemperatureError*, then it determines that class *TemperatureError* contains an attribute with a name (i.e., *isDetected*) that best matches the name of the term tagged as *A2* (i.e., *detected*).

To identify the attribute (or association) that best matches a phrase (Item 3), *OCLgen* first preprocesses the given phrase (i.e., removes spaces and preceding articles). Then it looks for an attribute (or association) whose name is a prefix or a postfix of the tagged phrase or, vice-versa, it looks for an attribute (or association) that starts or ends with the tagged phrase. For each matching attribute (or association), *OCLgen* computes a similarity score equal to the percentage of matching characters. When a matching attribute is not found, the process is repeated considering only the root of the given phrase (this allows to deal with plural names). *OCLgen* keeps the attribute with the best score.

For example, in the case of sentence S2 in Table IV, *OCLgen* adds the attribute *BodySense.itsNVM* to the list of lhs-variables because it terminates with *NVM*, the phrase tagged as A1 in sentence S2 (the score is 0.5 in this case).

To deal with classes introduced in the domain model to group attributes (Item 4), *OCLgen* traverses all the associations to classes related to the system class (or another class considered to identify the lhs-variable) and checks if these classes contain an attribute that best matches the phrase labelled with the role used to identify the attribute name (A1 or a support role). If a matching attribute is found then the attribute score is divided by the number of associations traversed. This compensates for the fact that the considered attribute might be loosely related to the class initially considered for the search (i.e., the system class or a class corresponding to the role A1).

In the case of sentence S1 in Table IV the lhs-variable *BodySense.itsOccupancyStatus.occupantClassForAirbagControl* is identified by traversing an association of the system class *BodySense*. In this case, the resulting score is equal to 0.5 because we have traversed one association (i.e., *OccupancyStatus*) and we have a perfect match between the attribute name and the noun phrase tagged as A1.

To deal with boolean attributes (Item 5), *OCLgen* further refines all the lhs-variables with a complex type (i.e., a class or a data type) by checking if they contain a boolean attribute that best matches one of the support roles appearing in the use case step (Lines 18 to 20 in Figure 6).

For example, in the case of sentence S2, *OCLgen* refines the variable *BodySense.itsNVM*, which was identified in previous steps. In this case, function *appendAdditionalAttributes* adds the variable *BodySense.itsNVM.isAccessible* to the list of lhs-variables because the class pointed by *itsNVM* contains a boolean attribute (i.e., *isAccessible*) with a name similar to the term tagged as *AM-PRD* (i.e., *accessible*).

The flexible approach implemented by *OCLgen* for the identification of lhs-variables leads to the identification of multiple lhs-variables for each use case step. Each lhs-variable has a score that is the average of the score of all the attributes/associations appearing in this variable (separated by dots). For example, in the case of sentence S2, function *findVariables* returns a list with three variables: *BodySense.itsNVM*, *NVM.isAccessible* and *BodySense.itsNVM.isAccessible* with scores 0.5, 0.83 and 0.66, respectively (details of the computation appear in Table VI).

VII. Identification of the term to be used in the right-hand side of the OCL expression

The rhs-term can be a literal, a variable, or a constant, and it captures some information about the value of the lhs-variable (e.g., it may indicate the value assigned to the lhs-variable after the 'execution' of the internal step of a use case specification).

OCLgen selects the rhs-term based on the type of the lhs-variable and on the support roles appearing in the sentence that were not already processed to select lhs-variables.

If the lhs-variable is of a boolean or numeric type, then the rhs-term is identified by translating phrases tagged with support roles into boolean or numerical form. For example, in the case of the verb *to set* the value is usually indicated by the term tagged as *A2*, which, in the case of the verb to set,

indicates the final state to which the term tagged as A1 has been set.

When the lhs-variable is a numeric type it often happens that the value on the right-hand side corresponds with a constant. To deal with these cases, when the word tagged with a support role cannot be translated to a boolean or numeric value, *OCLgen* looks for a constant in the domain model that best matches it.

Also, if the lhs-variable is a boolean and semantic role labeling indicates that the verb used in the use case step is negative, *OCLgen* negates the value to be used.

Instead, if the lhs-variable is of an enumeration type, *OCLgen* checks if the enumeration contains a value that best matches the phrases tagged with support roles. An example is sentence S4 in Table IV, where *BodySense.buildCheckStatus* is the lhs-variable in the OCL expression, which is of an enumeration type, *BuildCheckStatus*. Since this enumeration contains a term that matches the root of the verb appearing in S4 (i.e., *pass*), then the literal *BuildCheckStatus::Passed* is selected as rhs-term.

In the presence of boolean lhs-variables, or in certain transformation rules (e.g., *reset*), *OCLgen* uses default values when no phrase has been tagged with a support role, or when the word tagged with the support role has been used to determine the lhs-variable. This happens, for example, in the case of sentence S3, where the role A2 determines the name of the attribute, which is a boolean, and the default value *true* is thus used for the rhs-term.

OCLgen assigns a score also to the rhs-term. When the rhs-term is a boolean, numeric or enumeration literal the score is always set to one because in these cases *OCLgen* generates a rhs-term only in the presence of an exact match with the phrase tagged by the support role. When the rhs-term is a variable, the score is computed as in the case of the lhs-variable.

VIII. Identification of the OCL operators

The identification of the operator to be used in the OCL expression depends on the verb appearing in the sentence.

For most of the verbs we use the equal operator. For the verb *to be*, instead, we rely on the approach of Roy et al. [35]. This approach, for example, enables *OCLgen* to determine that the operator '$>$' should be used in the case of the sentence *"temperature is above 10"*.

For what it concerns the selection operator, instead, we have developed a solution that deals only with the pattern described in Figure 4, i.e., when the selection operator is used to exclude a subset of class instance types from a set. For example, in the case of sentence S5 in Table IV, the selection operator is used to indicate that the OCL expression holds for any error except memory errors and voltage errors.

OCLgen introduces a selection operator in the generated OCL constraint when the phrase tagged with A1 contains the keyword *except*. To identify the names of the class types to be excluded by the selection operation, *OCLgen* relies on the tags generated according to SRL NomBank. More precisely, it first identifies the adverbial clause that specifies which types should be excluded by the selection operator (this is done by looking for the phrases tagged as A2 by SRL) and then identifies all the distinct noun phrases within this clause (*voltage errors* and *memory errors* in the case of S5).

IX. Scoring of the generated constraints

The score associated with an OCL constraint should ideally measure both the completeness and correctness of the OCL constraints. Completeness relates to the extent to which all the concepts appearing in the sentence are accounted for. Correctness relates to how similar are the variable names in the OCL constraint to existing concepts in the use case specifications.

OCLgen measures the completeness of an OCL constraint in terms of the percentage of roles present in the use case sentence that are used to identify the terms of the constraint.

To compute the correctness of the lhs-variable and the rhs-term, instead, we use the following formula

$$correctness = \frac{(lhsScore + rhsScore + matchUniversalDeterminer)}{3}$$

where *lhsScore* and *rhsScore* are the similarity scores computed when identifying the lhs-variable and the rhs-term, respectively. *matchUniversalDeterminer* is a variable attempting to capture whether universal determiners (e.g., *any*, *every*, or *no*) are properly reflected in the constraint. If that is the case, the numerator of correctness is increased by one.

Properly processing universal determiners is important to derive precise constraints. For example, when applied to sentence S2 of Table IV, *OCLgen* may generate two valid OCL constraints, which are shown in the second and third line of Table VI. Ideally, in this case, we would like to prioritize the constraint on the third line because the use case step does not explicitely indicate that all the NVM components should be considered (we may have more than one).

We thus consider universal determiners to be properly reflected in a constraint in the following two cases: (1) When a universal determiner does not appear in the noun phrase tagged with A1 and the constraint refers to a specific instance associated with the system class (this is the case of sentence S2 above). (2) When a universal determiner appears in the noun phrase tagged with the role A1 and the constraint refers to all the instances of the type matching this phrase (this is the case of sentence S5 that checks if all the instances of class *Error* have the attribute *isDetected* set to false).

The score of an OCL constraint is computed as the average of the completeness and correctness score. Table VI shows the score of the OCL constraints generated by *OCLgen* for sentence S2 in Table IV.

X. Completeness and generalizability

The transformation rules integrated into *OCLgen* are verb-specific, which may limit the applicability of the approach because of the need to implement a considerable number of transformation rules. For example, the *Unified Verb Index* [36], which is a popular lexicon in the NLP community based on VerbNet and other lexicons, includes 8,537 English verbs.

TABLE VI
OCL CONSTRAINTS SCORING EXAMPLE FOR SENTENCE S2 OF TABLE IV

	Candidate OCL	Score
1	$BodySense.allInstances() \rightarrow forAll(i \mid i.itsNVM = ...)$	–
2	$NVM.allInstances() \rightarrow forAll(i \mid i.isAccessible = true)$	0.33
3	$BodySense.allInstances() \rightarrow forAll(i \mid i.itsNVM.isAccessible = true)$	0.94

Notes on the computed values:
The OCL constraint on the **first line** is ignored because incomplete (the attribute *BodySense.itsNVM* is a class type and does not enable the identification of any rhs-term). The constraint on the **second line** receives a score of 0.81, which results from a completeness score of 1 (all the roles are used to identify the constraint terms), and a correctness score of 0.61. The lhs-variable score is 0.83 and results from the division of the length of the word *accessible* (i.e., 10) by the length of the variable named *isAccessible* (i.e., 12). The score of the rhs-variable is equal to one, while the score for the universal determiner is zero because the constraint refers to all the instances of class NVM although no universal determiner is used in the use case step. The algorithm then computes $(0.83+1+0)/3 = 0.61$. In the case of the **third constraint**, the OCL score results from a completeness score of 1 (all the roles are used) and a correctness score of 0.87. The lhs-variable score is 0.66, which is the average between the score computed for attribute *itsNVM* (i.e., 0.5), and the score computed for the attribute *isAccessible* (i.e., 0.83). Then, since the score for the rhs-term is equal to 1 and the generated OCL refers to the system class, the correctness score is computed as $(0.66 + 1 + 1)/3 = 0.87$.

TABLE VII
VERBS UNLIKELY TO APPEAR IN USE CASE SPECIFICATIONS.

Reason 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 sounds / behaviours	bark, woof, quack
Communication verbs	tell, talk

We rely on two key solutions to make *OCLgen* scale and enable the processing of general use case specifications without the need for writing and testing hundreds of transformation rules. First, we rely on VerbNet classes to use a single transformation rule to target different verbs. Second, we have identified a subset of English verbs that are unlikely to appear in software requirements specifications, thus reducing the total amount of transformation rules that need to be implemented.

Since all the verbs appearing in a VerbNet class are guaranteed to have the same semantics (i.e., they appear in sentences with the same roles) we reuse the same transformation rule for all the verbs of a class that also happen to be synonyms according to WordNet.

To determine the subset of English verbs that should be targeted by *OCLgen* we have manually inspected all the classes of verbs appearing in VerbNet to determine if there exist classes that are unlikely to be used in software requirement specifications. We have identified nine main reasons to exclude verbs and we report them in Table VII along with some examples of the verbs excluded by our analysis. For example, we do not expect to find verbs describing human feelings. We excluded 225 VerbNet classes and 175 VerbNet subclasses. The results of our analysis are available online [37].

In addition to manually filtering out classes of verbs that are unlikely to appear in use case specifications we have further analyzed the remaining verbs to determine if the concepts expressed by those verbs are likely to be captured by OCL constraints that are generated by the meta-verb transformation rule. This further analysis shows that only 33 dedicated transformation rules are required to process the 87 classes of verbs likely to appear in requirements specifications. In our current implementation of *OCLgen* we have implemented seven transformation rules, including the meta-transformation rule, which enable the approach to handle 408 verbs.

XI. EMPIRICAL EVALUATION

We have performed an empirical evaluation of *OCLgen* aimed at addressing the following three research questions:

RQ1 Does *OCLgen* generate correct OCL constraints?

RQ2 To what extent does *OCLgen* support the automated generation of OCL constraints from use case specifications?

RQ3 What are the factors that limit *OCLgen* effectiveness?

A. Object of the study

The object of the study is *BodySense*™, an automotive embedded system developed by IEE [38], our industrial partner. To perform our study we have used the domain model, the use case specifications, and the OCL constraints that we used in previous work to generate test cases for *BodySense*™ [8].

B. Methods and Results

To respond to **RQ1** we have compared the OCL constraints generated by *OCLgen* with the OCL constraints manually written by *BodySense*™ engineers to adopt *UMTG*. All the OCL constraints for *UMTG* follow the pattern in Figure 4 and, for this reason, the comparison has been fully automated. In general, when processing a use case step, *OCLgen* can either generate a correct OCL constraint, a wrong OCL constraint, or it may not generate any result.

This research question aims to evaluate the precision of *OCLgen*. We compute the precision of *OCLgen* as the percentage of OCL constraints generated by *OCLgen* that are correct. *OCLgen* generates 69 OCL constraints in total, 66 of which are correct. The precision of *OCLgen* is thus 0.97, which is impressive since it shows that almost every time *OCLgen* generates a constraint, it generates the right one.

To respond to **RQ2** we measured the recall of *OCLgen* as the percentage of OCL constraints required by *UMTG* that were correctly generated by *OCLgen*.

OCLgen correctly generates 66 OCL constraints out of the 87 constraints required by *UMTG*, thus leading to a recall of 0.76. This value clearly indicates that the effort saved by *OCLgen* is significant since it correctly generates more than 75% of the constraints required by *UMTG*.

To respond to **RQ3** we have manually inspected the sentences not processed by *OCLgen*.

The main factor that affects the results generated by *OCLgen* is the presence of imprecise specifications. For example, the use case step *The system VALIDATES THAT the temperature is valid* does not provide enough information to understand what is a valid temperature (e.g., a temperature in a given range). Other cases where *OCLgen* fails are due to inconsistencies between the terminology used in use case specifications and in

the domain model. For example, the constraint expected for the use case step *the occupancy status is valid* (Table II), cannot be automatically derived 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*).

It is, however, possible to be more precise and consistent with domain modeling. For example, the notion *valid* can be defined as a derived attribute in the model. Such practice, which is standard with domain modeling [11], would make all or nearly all constraints processable by OCLGen.

C. Threats to validity

The main threat to validity in our study relates to generalizability, a common issue with industrial case studies. *OCLgen* might be tailored to *BodySense*™ (T1) and this system might not be representative (T2).

In Section IV, we have shown that the OCL patterns handled by *OCLgen* are expressive enough for embedded systems, thus partially addressing threat T1. Further, we have verified that *OCLgen* can generate constraints from sentences containing any of the patterns of the VerbNet classes.

To deal with threat T2, we have considered industrial use case specifications and OCL constraints that were developed in a representative automotive context, long before *OCLgen* was designed and developed.

XII. Related Work

Our discussion of related work addresses the automated generation of executable system test cases and OCL constraints from requirements written in natural language.

Approaches that automatically generate executable test cases from requirements written in natural language require that software specifications be written according to a controlled natural language (CNL) [5], [6]. The main difference between these approaches and *OCLgen*, is that the latter works in a specific context (use case specifications and domain modeling) but without relying on a constrained language. *OCLgen* has been conceived to work with *UMTG*, which requires that use case specifications be written according to RUCM, a format that imposes some restrictions on the use of natural language (e.g., it introduces some keywords). However, RUCM does not restrict the set of verbs or nouns that can be used in use case steps and thus does not limit the capability of engineers to describe the interactions between actors and the system. Furthermore RUCM keywords are used to specify input and output steps but do not constraint the writing of internal steps or validation steps (i.e., the ones processed by *OCLgen*). On the other hand, approaches using CNLs introduce stronger limitations, e.g., NAT2TEST [6] supports only seven verbs of the English language. Further, because these approaches are based on transformation rules that are verb specific, they are hard to extend. *OCLgen*, instead, relies on VerbNet classes to enable the processing of general use case specifications with a limited set of transformation rules.

Other works [39], [20], [21] focus on the generation of class invariants and method pre- and post-conditions, from requirements written in natural language, which, in principle, could be used for the purpose of testing. Pandita *et al.* focus on API's description [39] written according to a CNL and thus cannot process general requirements specifications. NL2OCL [20] and NL2Alloy [21], instead, process a UML class diagram and requirements in natural language to derive class invariants and method pre- and post- conditions. These 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 (e.g., an attribute is a characteristic), and rely on the presence of specific keywords to determine passive voices or identify the operators to be used in constraints. The generation of constraints is rule-based and the authors do not provide a solution to ease the processing of a potentially large number of verbs with a reasonable number of transformation rules. Thanks to the meta-verb rule and the exploitation of VerbNet classes, in contrast, *OCLgen* can process a large set of verbs with few transformation rules.

Though NL2OCL [20] and NL2Alloy [21] are no longer available for comparison, they seem more useful for deriving class invariants including simple comparison operators (which was the focus of the empirical evaluation in [20]), rather than for generating pre- and post-conditions (which is the focus of *OCLgen*). Pre- and post-conditions are necessary for deriving test inputs, while class invariants are not. This is the reason why in this paper we focus on testing and leave the analysis of other application contexts of *OCLgen* for future work.

XIII. Conclusion

In this paper we presented *OCLgen*, an approach that automatically generates OCL constraints that capture the pre- and post-conditions of the steps appearing in use case specifications written in natural language. The approach complements *UMTG*, a technique that automatically generates an executable test suite that covers all the use case scenarios appearing in use case specifications. More specifically, *OCLgen* automates the generation of OCL constraints, from their natural language descriptions, that are required by *UMTG* to derive test inputs. In *UMTG*, OCL constraints are manually provided by engineers.

Empirical results obtained with the use case specifications of an automotive system show that *OCLgen* automatically generates more than 75% of the OCL constraints required by *UMTG*. Furthermore, we observe that more than 95% of the constraints generated by *OCLgen* are correct. These results show that *OCLgen* is a promising step towards the fully automated generation of system test cases from natural language use case specifications. A manual analysis shows that the cases where *OCLgen* fails are due to imprecise and incomplete natural language specifications.

Acknowledgment

Supported by an IEE grant and by the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement number 694277). We would like to thank Thierry Stephany and the IEE software engineers for their support.

REFERENCES

[1] ISO, "ISO-26262: Road vehicles – functional safety."

[2] 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 *Proceedings of International Conference on System Analysis and Modeling: Models and Reusability*, 2014, pp. 142–157.

[3] E. Sarmiento, J. C. S. d. P. Leite, and E. Almentero, "C&L: Generating model based test cases from natural language requirements descriptions," in *International Workshop on Requirements Engineering and Testing*, 2014, pp. 32–38.

[4] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066116301153

[5] 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.

[6] 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, no. P3, pp. 275–297, Dec. 2014. [Online]. Available: http://dx.doi.org.proxy.bnl.lu/10.1016/j.scico.2014.06.007

[7] A. Wyner, K. Angelov, G. Barzdins, D. Damljanovic, B. Davis, N. Fuchs, S. Hoefler, K. Jones, K. Kaljurand, T. Kuhn, M. Luts, J. Pool, M. Rosner, R. Schwitter, and J. Sowa, "On controlled natural languages: Properties and prospects," in *Proceedings of the 2009 Conference on Controlled Natural Language*, ser. CNL'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 281–289. [Online]. Available: http://dl.acm.org/citation.cfm?id=1893475.1893495

[8] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2015, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771812

[9] 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 *International Conference on Software Testing, Verification, and Validation*, 2008, pp. 82–91.

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

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

[12] 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. [Online]. Available: http://cogcomp.cs.illinois.edu/papers/PunyakanokRoYi07.pdf

[13] G. A. Miller, "WordNet: A Lexical Database for English," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/219717.219748

[14] M. Palmer, "SemLink: Linking PropBank, VerbNet and FrameNet." in *Proceedings of the Generative Lexicon Conference*, Pisa, Italy, Sept 2009.

[15] 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.

[16] 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/.

[17] B. Zhang, E. Hill, and J. Clause, "Automatically generating test templates from test names (n)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 506–511. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.68

[18] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931061

[19] 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 *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 327–336.

[20] I. S. Bajwa, B. Bordbar, and M. G. Lee, "Ocl constraints generation from natural language specification," in *IEEE International Enterprise Distributed Object Computing Conference*, Oct 2010, pp. 204–213.

[21] I. S. Bajwa, B. Bordbar, K. Anastasakis, and M. Lee, "On a chain of transformations for generating alloy from nl constraints," in *International Conference on Digital Information Management*, Aug 2012, pp. 93–98.

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

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

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

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

[26] M. Palmer, D. Gildea, and P. Kingsbury, "The proposition bank: An annotated corpus of semantic roles," *Comput. Linguist.*, vol. 31, no. 1, pp. 71–106, Mar. 2005. [Online]. Available: http://dx.doi.org/10.1162/0891201053630264

[27] A. Meyers, R. Reeves, C. Macleod, R. Szekely, V. Zielinska, B. Young, and R. Grishman, "The nombank project: An interim report," in *HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, A. Meyers, Ed. Boston, Massachusetts, USA: Association for Computational Linguistics, May 2 - May 7 2004, pp. 24–31.

[28] M. Gerber, J. Chai, and A. Meyers, "The Role of Implicit Argumentation in Nominal SRL," in *Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the ACL (NAACL HLT)*, 2009, pp. 146–154.

[29] University of Illinois, "CogComp NLP pipeline demo," 2017. [Online]. Available: http://cogcomp.org/page/demo_view/SRL

[30] K. Kipper, H. T. Dang, and M. Palmer, "Class-based construction of a verb lexicon," in *Proceedings of National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 2000, pp. 691–696. [Online]. Available: http://dl.acm.org/citation.cfm?id=647288.721573

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

[32] G. A. Miller, "WordNet: A Lexical Database for English," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/219717.219748

[33] H. F. Ledgard, "A human engineered variant of bnf," *SIGPLAN Not.*, vol. 15, no. 10, pp. 57–62, Oct. 1980. [Online]. Available: http://doi.acm.org.proxy.bnl.lu/10.1145/947727.947732

[34] 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, Mar. 1970.

[35] S. Roy, T. Vieira, and D. Roth, "Reasoning about quantities in natural language," *Transactions of the Association for Computational Linguistics (TACL)*, vol. 3, 2015. [Online]. Available: http://cogcomp.org/papers/RoyViRo15.pdf

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

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

[38] "IEE sensing solutions," http://www.iee.lu.

[39] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *Proceedings of International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 815–825. [Online]. Available: http://dl.acm.org.proxy.bnl.lu/citation.cfm?id=2337223.2337319