UMTG: A Toolset to Automatically Generate System Test Cases from Use Case Specifications

Chunhui Wang[†], Fabrizio Pastore[†], Arda Goknil[†], Lionel C. Briand[†], Zohaib Iqbal[†‡]
[†] Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
[‡] Quest Lab, National University of Computer & Emerging Sciences (FAST NU), Islamabad, Pakistan
{chunhui.wang,fabrizio.pastore,arda.goknil,lionel.briand}@uni.lu zohaib.iqbal@nu.edu.pk

ABSTRACT

We present UMTG, a toolset for automatically generating executable and traceable system test cases from use case specifications. UMTG employs Natural Language Processing (NLP), a restricted form of use case specifications, and constraint solving. Use cases are expected to follow a template with restriction rules that reduce imprecision and enable NLP. NLP is used to capture the control flow implicitly described in use case specifications. Finally, to generate test input, constraint solving is applied to OCL constraints referring to the domain model of the system. UMTG is integrated with two tools that are widely adopted in industry, IBM Doors and Rhapsody. UMTG has been successfully evaluated on an industrial case study.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

NLP, Use Case Specifications, Test Cases Generation

1. INTRODUCTION

To ensure traceability between requirements and system test cases in safety critical domains, e.g. automotive, system test cases are often manually derived from functional requirements written in natural language. As a result, the definition of system test cases is time-consuming and challenging under time constraints.

Considerable research has been devoted to automatically deriving test cases from requirements in natural language [12]. Most of the existing approaches require either additional behavioral modeling (e.g., activity diagrams [13]) or manual intervention by the testers, for example, to manually derive test inputs [10]. Commercial model-based testing (MBT) tools such as BPM-X [1], SpecExplorer [8], or Conformiq Designer [2] automatically generate system test cases by using behavioral models, e.g., BPMN [14] and finite state machines. In modern industrial systems, behavioral models tend to be complex and expensive. Our experience with our industry partner IEE [7] shows that the adoption of behavioral modeling, at the required level of detail for automated testing, is not practical for system test automation [16].

In this paper, we present *UMTG*, a toolset that generates executable system test cases by exploiting the behavioural information implicitly described in use case specifications. UMTG automates the approach detailed in [16]. UMTG requires a domain model of the system, which enables the definition of constraints that are used to generate test data.

UMTG applies Natural Language Processing (NLP) on use case specifications to identify domain entities and constraints, i.e. the pre-, post- and guard- conditions listed in the use cases. This information is used to build *Use Case Test Models (UCTMs)* that capture the control flow implicitly described in the use case specification.

To generate test data, UMTG expects constraints, referring to the domain model, to be defined with the Object Constraint Language (OCL) [15] since OCL is the natural choice when defining high-level constraints on class diagrams. In order to generate test inputs, UMTG relies upon a constraint solver [9] which processes the domain model and associated OCL constraints given by the software engineer.

We applied UMTG on the use case specifications of *BodySense*, an automotive sensor system provided by IEE [7]. The results show the effectiveness and scalability of UMTG that generates test cases that cover more execution scenarios than those covered by manual test suites [16].

This paper proceeds as follow: Section 2 presents the workflow of our toolset while Section 3 gives an overview of the toolset architecture. Section 4 presents example test cases generated by UMTG, Section 5 summarizes the results achieved with an industrial case study, while Section 6 concludes the paper.

2. UMTG WORKFLOW

Figure 1 shows the activities performed by software engineers to automatically generate system test cases with UMTG.

In Step 1, the software engineer writes use case specifications according to Restricted Use Case Modeling (RUCM). RUCM [17] is a restricted use case format with a set of keywords and restriction rules that enable natural language

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803187

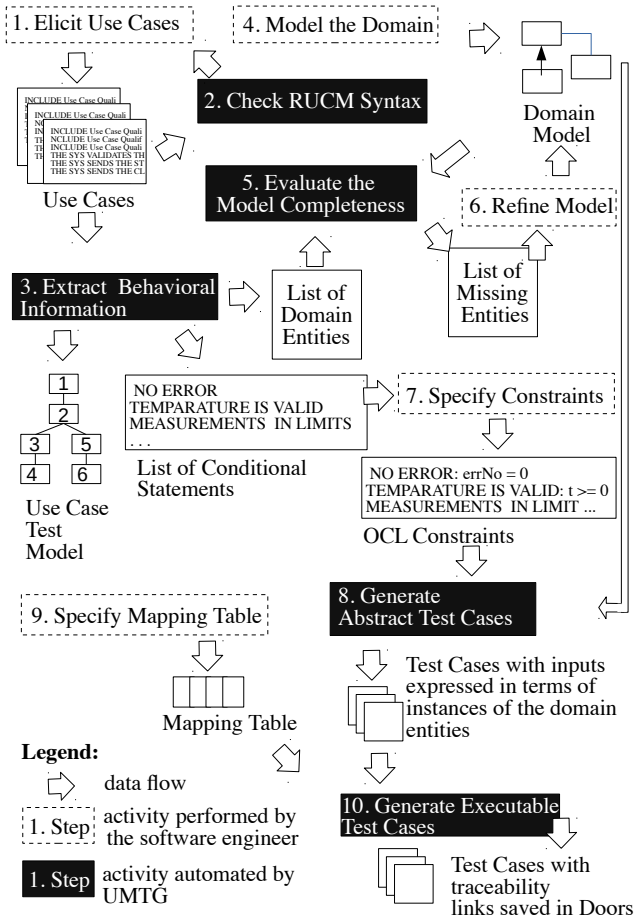


Figure 1: The UMTG workflow

processing of use cases. Figure 2 shows an excerpt of a use case specification of BodySense with basic and alternative flows. In Figure 2, capital letters are used for RUCM keywords. For example, the use case step in line 8241 includes the keyword *VALIDATES THAT*. According to RUCM, this keyword must be followed by a conditional statement. In line 8241 of Figure 2, the conditional statement is *"the occupant class for airbag control is valid and the occupant class for seat belt reminder is valid"*. The next use case step, i.e., line 8242 in Figure 2, is taken only if the condition holds, otherwise, the alternative flow starting from line 8257 is selected (see [16] for a detailed description of RUCM).

We assume use cases are managed with IBM Doors [5] which does not put any restriction on the structure of use cases, thus allowing engineers to adopt the RUCM template.

In Step 2, UMTG checks if the use case specifications conform to the RUCM template. If not, software engineers are expected to correct the use case specifications.

In Step 3, UMTG extracts behavioural information from the use case specifications by means of NLP. During this step, UMTG generates three different outputs: a list of domain entities appearing in the use case specifications, a list of conditional statements in the use case specifications, and the Use Case Test Model. The Use Case Test Model makes the implicit control flow in a use case specification explicit. It is an intermediate model that is not meant to be inspected by end-users but is used to drive test generation.

ID	
8224	**1 UCS: BodySense III AUDI MLB evo - Normal operation**
8225	**1.1 Use Case: Identify Initial Occupancy Status of a Seat**
8228	**1.1.1 Precondition**
8229	The system has been initialized.
8238	**1.1.2 Basic Flow**
8239	1. The seat SENDS occupancy status TO the system.
8240	2. INCLUDE USE CASE Classify occupancy status.
8241	3. The system VALIDATES THAT the occupant class for airbag control is valid and the occupant class for seat belt reminder is valid.
8242	4. The system SENDS the occupant class for airbag control TO AirbagControlUnit.
8243	5. The system SENDS the occupant class for seat belt reminder TO SeatBeltControlUnit.
8246	Postcondition: The occupant class for airbag control and the occupant class for seat belt reminder have been sent.
8257	**1.1.4 Specific Alternative Flow**
8258	RFS 3
8259	1. IF the occupant class for airbag control is not valid and the occupant class for seat belt reminder is not valid THEN
8260	2. The system SENDS the previous occupant class for airbag control TO AirbagControlUnit.
8261	3. The system SENDS the previous occupant class for seat belt reminder TO SeatBeltControlUnit.
8262	4. RESUME STEP 1.
8263	5. ENDIF
8265	Postcondition: The previous occupant classes for airbag control and seat belt reminder has been sent to AirbagControlUnit and to SeatBeltControlUnit respectively.

Figure 2: RUCM Use Case Specification in IBM DOORS

In Step 4, the software engineer designs the domain model of the system using IBM Rational Rhapsody. In Step 5, UMTG checks the model completeness by verifying if all the domain entities identified in the use case specifications are present in the domain model. If there are missing domain entities, the software engineer is expected to refine the domain model accordingly (Step 6).

In Step 7, the software engineer reformulates the conditional statements as OCL constraints referring to the domain model. Figure 3 shows the OCL constraint for the conditional statement that specifies that *the occupant class for airbag control is valid* when the value of attribute *occupantClassForSeatBeltReminder* is different from *Error*.

$$inv\ TheOccupantClassForAirbagControlIsValid : BodySenseSystem.allInstances() \rightarrow forAll(b|b.occupancyStatus.occupantClassForSeatBeltReminder <> OccupantClass :: Error)$$

Figure 3: OCL constraint for the conditional statement *the occupant class for airbag control is valid*.

In Step 8, UMTG generates abstract test cases. In this step UMTG identifies the test inputs required to cover each scenario of the use case by solving the path conditions of the Use Case Test Model. We use the term *use case scenario* for a sequence of use case steps that start with a use case precondition and ends with a postcondition of either a basic or alternative flow. UMTG generates test inputs that cover all paths in the Use Case Test Model and therefore all possible use case scenarios. Since the test inputs generated in this step are expressed in terms of instances of the entities in the domain model and cannot directly be executed on the target platform, we refer to such inputs as *abstract test cases*.

To generate executable test cases, abstract test inputs must be translated into concrete test inputs, e.g. byte vectors. To this end the software engineer provides a mapping table with regular expressions that map abstract inputs to invocations of test driver functions including concrete inputs (Step 9).

In Step 10, UMTG generates executable test cases by applying the mapping table on the abstract test cases. The generated test cases are automatically loaded into IBM Doors and traceability links are created.

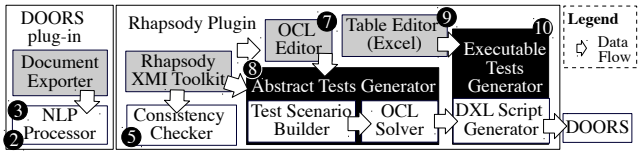
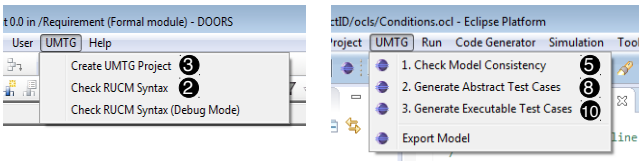


Figure 4: UMTG architecture (gray boxes show third party components, black boxes UMTG components with nested components)



IBM Doors plug-in IBM Rhapsody plug-in

Figure 5: Menus provided by the UMTG plug-ins

3. UMTG ARCHITECTURE

Figure 4 shows the UMTG architecture. To ease its industrial adoption, we integrated UMTG with two widely used tools that are adopted by our industry partner: IBM Doors [5] and IBM Rhapsody [6] (UMTG works with the version of Rhapsody integrated with the Eclipse development environment [3]).

The main components of UMTG are two plug-ins that extend Doors and Rhapsody. These plug-ins provide the user interface of UMTG and orchestrate the other components of UMTG that implement the steps in Figure 1 (see Figure 4 where the black circles denote the steps).

The UMTG Doors plug-in provides the menu buttons that activate the UMTG steps related to the elicitation of use cases while the UMTG Rhapsody plug-in provides the menu buttons that activate the steps related to test generation. Figure 5 shows the contextual menus provided by the UMTG plug-ins where black circles denote the corresponding steps in Figure 1.

The UMTG Rhapsody plug-in also provides the interface to visualize and edit the artefacts produced by UMTG. UMTG takes advantage of the plug-in architecture of Eclipse to reuse functionality provided by third party plug-ins. In particular, UMTG relies upon the following user interfaces: the *Eclipse Web Browser* to visualize the domain entities missing from the domain model, the *OCL editor* provided by the Eclipse OCL plug-in [4] to support software engineers in editing the list of OCL constraints, the *Eclipse Text Editor* to visualize abstract test cases, the default *Eclipse Table Editor* (e.g. Microsoft Excel) to edit mapping tables, and the *Eclipse Project Explorer* to list the UMTG artefacts.

The other components in Figure 4 implement the steps of the UMTG workflow automated by the UMTG toolset, i.e. Steps 2, 3, 5, 8, and 10 in Figure 1. The *NLP Processor* implements Steps 2 and 3. It is based on the GATE workbench [11], an open source NLP framework. To load use cases from IBM Doors, UMTG uses the *Doors Document Exporter*, an API that exports Doors content as text files.

The UMTG Rhapsody plug-in implements steps 5, 8, and 10. The UMTG Rhapsody plug-in relies upon a third party application, the *Rhapsody XMI Toolkit*, to export the domain model in the XMI format. This is necessary since Rhapsody saves models into its own proprietary format. The *Consistency Checker* and the *Abstract Tests Generator* implement steps 5 and 8. The *OCL Solver* is the constraint solver described in [9].

The *Executable Test Generator* implements Step 10, it takes the mapping table and the abstract test cases as input, and generates the executable test cases as output. The *Executable Test Generator* exploits the Door eXtension Language (DXL) to load the generated test cases into Doors. The DXL functionality is also used to automatically generate the traceability links between use case specifications and the generated test cases. UMTG adds to each generated test case a set of traceability links indicating the flow of the use case specifications covered by the test cases. Furthermore, for each use case flow covered by a test case, it generates traceability links indicating the test cases covering it.

4. TEST CASES AND MAPPING TABLES

UMTG generates two sets of test cases: abstract test cases and executable test cases (see Figure 6).

Abstract test cases include test inputs and oracles expressed in terms of domain model entities. The left part of Figure 6 shows an example of an abstract test case. Abstract test cases are saved in text files under the Eclipse/Rhapsody workspace. Each abstract test case includes a header section that depicts the steps of the scenario under test, and a body section with a list of input operations and oracles. The header lines in Figure 6 show that the test case covers a scenario in which the condition *the cable shield integrity is valid* is false (header lines begin with the symbol #). One of the test inputs in Figure 6 is an assignment of the value *NotOK* to *CableShieldIntegrityStatus* (this is the value that falsifies the condition above).

The executable test cases generated by UMTG extend the abstract test cases by including calls to the test driver functions that need to be invoked to execute the system.

The right part of Figure 6 shows a portion of an executable test case generated by UMTG. The generated test case includes lines with abstract test inputs that are included to provide high-level operation descriptions. These lines are followed by the test driver functions with the concrete inputs to be used to test the system. Label A in Figure 6 points to the high level operation description that indicates that the test case must set the *CableShieldIntegrityStatus* to the 'failed' status. Label B points to the corresponding test driver function, i.e., *Set AMGB*. *Set AMGB* is used to simulate an input signal coming from a sensor (in this case it sends an error status on the channel of the *shield integrity* sensor). The generated test case also contains test oracles. Label C shows an oracle implemented by means of an invocation of function *Lin Publish*, which is used to check the signals sent on a channel.

Executable test cases are generated by means of a mapping table that is provided by software engineers. Figure 7 shows a portion of the mapping table used for *BodySense*. To generate calls to driver functions, UMTG parses each line in the abstract test cases according to the mapping table. The mapping table is made of five columns. The first two columns provide operation names and regular expressions that match an input in the abstract test case. The last three columns provide the driver function calls and parameters that should be added to the executable test case when an abstract input matches the regular expression.

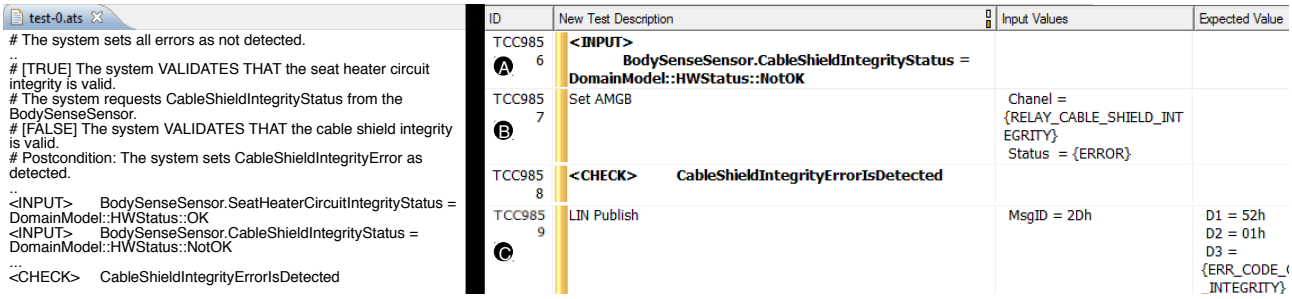


Figure 6: Example of an Abstract (left) and a corresponding Executable (right) Test Case Generated by UMTG

1	<INPUT>	BodySenseSensor.CableShieldIntegrityStatus = DomainModel::HWStatus::NotOK	Set AMGB	Channel = {RELAY_CABLE_SHIELD_INTEGRITY} Status = {ERROR}	
2	<CHECK>	CableShieldIntegrityErrorIsDetected	LIN Publish	MsgID=2Dh	D1=52h D2=01h D3={ERR_C

Figure 7: Portion of a Mapping Table

5. EVALUATION

UMTG has been adopted to automatically generate test cases from the use case specifications of BodySense [16]. The case study includes six different use case specifications of varying length. Each specification includes from 25 to 50 steps and several alternative flows, from 6 to 13. While the domain model includes 58 entities, 63 attributes, 22 associations and 35 inheritance relations.

Since both the elicitation of use case specifications and the definition of a domain model are common software engineering practices, to evaluate the additional modelling effort required by UMTG we focus on the number of OCL constraints to be defined by software engineers. We defined 53 OCL constraints for BodySense, a number which has been considered acceptable by IEE engineers.

In the case study, UMTG covers all the scenarios of the use cases under test, more than the manually derived test cases. UMTG covers a total of 100 scenarios, while manually written test cases cover 80 scenarios. These results mostly depend on the fact that RUCM reduces imprecision and incompleteness in use cases, thus leading to the identification of more scenarios.

Our evaluation also shows that UMTG effectively generates test inputs from use case specifications. The tool scales as the entire system test generation can easily be run over night for a representative industrial system.

6. CONCLUSION

We presented our tool, UMTG, for automatically generating executable system test cases by exploiting the behavioural information implicitly described in use case specifications. UMTG successfully brings automatic test generation to industrial practice by integrating advanced technologies, i.e. NLP and OCL constraint solving, with state-of-practice development environments such as Eclipse, IBM Doors, and Rhapsody.

Empirical results obtained with an industrial case study show that UMTG is effective to automatically generate test inputs from use case specifications. Our experience indicates that modeling requirements in UMTG is realistic and feasible in an industrial context. In the future, we plan to conduct more case studies with more involvement of the IEE engineers to further evaluate the usability of our tool.

UMTG and a video demonstration are available at the following URL: https://sites.google.com/site/umtgtestgen.

7. ACKNOWLEDGMENT

This work has been supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03), and by IEE. We would like to thank Thierry Stephany for his support.

8. REFERENCES

[1] Bpm-x. http://www.bpm-x.com.
[2] Conformiq designer. http://www.conformiq.com/.
[3] Eclipse IDE. http://www.eclipse.org.
[4] Eclipse OCL. http://www.eclipse.org/modeling/.
[5] IBM Doors. http://www-03.ibm.com/software/products/en/ratidoor.
[6] IBM Rhapsody. http://www-03.ibm.com/software/products/en/ratirhapfami.
[7] IEE sensing solutions. http://www.iee.lu.
[8] SpecExplorer. http://research.microsoft.com/en-us/projects/specexplorer/.
[9] S. Ali, M. Zohaib Iqbal, A. Arcuri, and L. Briand. Generating test data from OCL constraints with search techniques. *IEEE TSE*, 39(10):1376–1402, 2013.
[10] A. Bertolino and S. Gnesi. Use case-based testing of product lines. *SIGSOFT SEN*, 28(5):355–358, 2003.
[11] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *In ACL'02*, 2002.
[12] 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. *JSS*, 84(8):1379–1393, 2011.
[13] B. Hasling, H. Goetz, and K. Beetz. Model based testing of system requirements using UML use case models. In *ICST 2008*. IEEE.
[14] OMG. BPMN specification. http://www.bpmn.org/.
[15] OMG. The Object Constraint Language. http://www.omg.org/spec/OCL/.
[16] C. Wang, F. Pastore, A. Goknil, L. Briand, and M. Z. Iqbal. Automatic generation of system test cases from use case specifications. In *ISSTA 2015*. ACM.
[17] T. Yue, L. C. Briand, and Y. Labiche. Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM TOSEM*, 22(1), 2013.