20 e 21 de outubro
Instituto Nacional de Pesquisas Espaciais - INPE
São José dos Campos - SP

Model-Based Testing Considering Natural Language Requirements

Valdivino Alexandre de Santiago Júnior[1], **Nandamudi L. Vijaykumar**[1,2], **José Demísio S. da Silva**[1,2]

[1]Programa de Pós-Graduação em Computação Aplicada
Instituto Nacional de Pesquisas Espaciais (INPE)
Av. dos Astronautas, 1758 – 12227-010 – São José dos Campos – SP – Brazil

[2]Laboratório Associado de Computação e Matemática Aplicada (LAC)
Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos – SP – Brazil

valdivino@das.inpe.br, {vijay,demisio}@lac.inpe.br

Abstract. *A methodology addressing test case generation based on Natural Language (NL) requirements specifications is presented in this paper. A tool that supports the methodology translates automatically NL requirements into models. Once the models are obtained, another tool, GTSC, is used to generate the test cases. The methodology identifies scenarios for testing by means of combinatorial designs, hence providing a formal manner to derive scenarios. Our methodology is easy to use and at the same time is supported by a formal method, consequently leading to potential decrease of time with respect to test case generation.*

Keywords: *Model-Based Testing, Natural Language Requirements, Formal Methods.*

1. Introduction

Model-Based Testing (MBT) is as a type of testing in which tests are derived from software behavioral models [El-Far and Whittaker 2001]. This definition includes formal model/language specifications and other notations, like Unified Modeling Language (UML) models [OMG 2007]. Among the formal methods used for system and acceptance model-based test case generation are Statecharts [Harel 1987] [Santiago et al. 2008], Finite State Machines (FSMs) [Sidhu and Leung 1989], and Z [Cristiá et al. 2010].

Taking into account model-based system and acceptance test case generation, a test designer usually breaks down the entire system based on ways to use the system (scenarios), and then models are derived to address each scenario. Based on such models, test cases can be obtained. One problem here is the difficulty and the time necessary to

identify the scenarios and consequently to generate the test cases. The test designer tries to identify the scenarios based on the very first deliverables elaborated within the software development lifecycle, such as software requirements specifications.

The academic community has been advocating the use of formal methods to elaborate software requirements specifications, but they require high expertise for that. Thus, formal methods are not widely used in industrial practice. The conclusion is that Natural Language (NL) is still the most used to elaborate requirements specifications [Mich et al. 2004] as it is the simplest way for stakeholders. Moreover, NL may be associated to requirements modeling methods, like use case models where a textual description exists in order to narrate the behavior through a sequence of actor-system interactions.

The identification of scenarios, their respective models, and generation of test cases based on NL requirements documents are cumbersome and time-consuming tasks specially for real complex applications. This paper thus presents a methodology which aims model-based test case generation considering NL requirements specifications. The methodology is supported by a tool that makes it possible to automatically translate NL requirements into Statecharts models. Once the Statecharts are derived, the *Geração Automática de Casos de Teste Baseada em Statecharts* (GTSC - Automated Test Case Generation based on Statecharts) environment [Santiago et al. 2008] is used to generate the test cases. Our methodology uses combinatorial designs [Mathur 2008] to identify the scenarios for system and acceptance testing. The key benefits from applying our methodology/tool within a test process are the easiness of use, and at the same time based on the support of a formal method, consequently leading to potential decrease of time with respect to test case generation.

This paper is organized as follows. Section 2 presents our methodology. Section 3 describes the computational aspects of the activity within the methodology that generates the model. Section 4 presents the application of our methodology/tool using as case study a space application software product. Conclusions and future directions are in section 5.

2. A test case generation methodology

Creating a Dictionary is the first activity that the test designer shall accomplish in order to apply our methodology. The Dictionary defines the application domain and it is composed of three sets: $Names$ that will mainly define the names of the states of the model; $Reactiveness$ which is composed of two subsets ($input\_event$ and $output\_event$) that represent the Reactiveness of the system; and $Control$ that characterizes specific control behaviors.

The $Names$ and $Reactiveness$ sets shall be defined by the user, but the $Control$ set is already defined within the tool that supports the methodology, although the user can change it if needed. It is not required from the user any knowledge in formal methods and their respective notations to define the application domain. The Reactiveness feature of the Dictionary comes into picture because the main systems that our methodology aims to address are the reactive ones.

After the definition of the Dictionary, the scenarios shall be identified. Our methodology uses combinatorial designs, a set of techniques for test case generation

[Mathur 2008], to this end. However, the methodology uses combinatorial designs not to generate test cases but rather to identify scenarios. The basic idea is to identify *factors* (input variables) and *levels* (values assignable to a factor) and to use a combinatorial designs algorithm to determine the set of levels, one for each factor, known as a *factor combination* or *run*[1].

Once the factor combinations are generated, the test designer shall interpret each one in order to define the scenarios. Each factor combination will derive a scenario. Hence, the test designer is able to select a set of requirements (a scenario) and he/she can input these NL requirements via Graphical User Interface (GUI) of the tool that supports the methodology. Thus, our methodology provides a formal manner to identify scenarios for system and acceptance test case generation, instead of adopting the usual ad hoc approach.

The next activity within the methodology is the generation of the Statecharts model. This activity will be discussed in detail in section 3. After that, test cases can be generated by using the GTSC environment [Santiago et al. 2008]. GTSC allows test designers to model software behavior using Statecharts and/or FSMs in order to automatically generate test cases based on some test criteria for FSM and some for Statecharts. At present, GTSC has implemented Unique Input/Output (UIO), Distinguishing Sequence (DS) [Sidhu and Leung 1989] and H-switch cover [Souza 2010] test criteria for FSM models, and three test criteria from the *Statechart Coverage Criteria Family* (SCCF) [Souza 2000], all-transitions, all-simple-paths and all-paths-k-C0-configuration, targeting Statecharts models. In other words, test criteria define the rules that drive test case generation in GTSC.

In order to use GTSC, a user shall translate the Statecharts behavioral model into an XML-based language named *PerformCharts Markup Language* (PcML) [Santiago et al. 2006]. In the case of our tool, the idea is to automatically translate the generated model into the PcML language. Based on a PcML document, a flat FSM is generated by GTSC[2]. This flat FSM is indeed the basis for test case generation.

Having created the test cases for a particular scenario, the test designer may start again inserting the NL requirements for the next scenario. This process shall be repeated until there is no more scenario left.

3. Generation of the model

When the user selects the options in the GUI of our tool in order to generate the model, a set of algorithms take place to meet this goal. The first task refers to the generation of *Behavior-Subject-Action-Object* (BSAO) 4-tuples. The BSAO tuples are an extension of the concept of SAO triads used in the *Java Requirement Analyzer* (J-RAn) tool [Fantechi and Spinicci 2005]. J-RAn implements a Content Analysis technique to support the analysis of inconsistency and incompleteness in NL requirements specifications.

[1]The technique adopted within the methodology is the *Mixed-Level Covering Array* (MCA) which allows factors to assume levels from different sets. The algorithm used is the *In-Parameter-Order* (IPO), a procedure that can generate MCAs.

[2]A flat FSM is a model where all hierarchical and orthogonal features of a Statecharts model were removed. PerformCharts tool [Vijaykumar et al. 2006], one of the components of the GTSC environment, is responsible for that.

Based on the NL document, this technique explores the extraction of the interactions between the entities described in the specification as SAO triads. These SAO triads are obtained with the help of the *Link Grammar Parser* [Sleator and Temperley 1993], a syntactic parser of English based on *Link Grammar*, a formal grammatical system.

In our tool, the first extension is the inclusion of Behavior features ("B") in the SAO triad so that it will be transformed into a BSAO 4-tuple. The reasoning behind this relies on the fact that words like *if* determine a particular behavior in the generated model. For instance, finding an if-then-else situation in one or several NL requirements (e.g. in one requirement: "If the system works according to ..."; in the same or in the next requirement: "On the other hand, if the system does not work according to ...") may imply that the behavioral model will have a state with two outgoing transitions each one representing the possible outcome of the if-then-else situation.

The second modification is related to the object identification. J-RAn presented a huge number of missing extractions [Fantechi and Spinicci 2005], and one possible explanation for this fact was because J-RAn used a single link type ("O") of the *Link Grammar Parser* to identify objects. However, it is possible that there is no explicit object generated by *Link Grammar* depending on the NL requirement. Consider the requirement below:

Users' data shall be updated on the server every 12 hours.

By using *Link Grammar*, there is no object because none of the link types regarding object ("O", "OT", ...) appears in the parser output. We tried to overcome situations like this in the tool that supports our methodology. We developed and implemented an algorithm to automatically identify the BSAO tuples. The algorithm makes use of the *Stanford Part-Of-Speech (POS) Tagger* [Toutanova et al. 2003] in order to identify the lexical categories (i.e. the parts of speech) of each sentence of the NL requirements. Actually, after the user has defined the NL requirements that form a scenario, we combine all such requirements into a file. This file is input to the *Stanford POS Tagger* which assigns the POS of each word (e.g. noun, verb, adjective, ...). Thus, the algorithm first verifies if there is a preposition or subordinating conjunction and also if such word is in the $Control$ set. If these conditions are matched, then the B element of the tuple is assigned to such word. If not, B is empty.

After the determination of B, the Subject (S), Action (A), and Object (O) elements are identified. Essentially, what the agorithm does is to verify whether the POS tags of words are equal to predefined POS tags that characterize a Subject, an Action, or an Object. For S and O, the selected POS tags are commom nouns (singular and plural), proper nouns (singular and plural), and adjectives. For A, the POS tags are verbs and adverbs. If they match, the corresponding S, A, and O elements of the tuple are fulfilled. However, a BSAO tuple is created if and only if a Subject and an Action and an Object were determined. If any of these elements was not determined, no tuple is created. This is to avoid several situations which might occur in NL sentences, and whose might produce ill-formed tuples. For instance, a piece of a sentence might derive an S, an A but not an O because the sentence has ended. In this way, a BSAO tuple in not generated.

After the generation of the tuples, the model shall be created. Several algorithms were developed and implemented in order to reach this goal. In our tool, the generated

model is a set whose each element represents a transition in the model. Hence, each element has these four fields: source state, input event, output event, and destination state.

The initial idea is to denote the states of the model with the Subject component of the BSAO tuple. However, an algorithm wonders whether there is already a source state in the model with the same name of the current BSAO Subject. The algorithm will return a name for the state just adding an underscore followed by an incrementing number after the Subject, if there is already a same state name in the model; otherwise, it will return the same BSAO Subject to be assigned as the name of the source state.

The Reactiveness set of the Dictionary plays an important role on defining the input and output events within a transition. If the Object of the BSAO tuple exists in the $input\_event$ subset, then the input event will be assigned to such value, and the output event will have the value of the corresponding element of the $output\_event$ subset of the $Reactiveness$ set. However, if the tuple Object does not match any value in $input\_event$, the input event will be formed by a combination of Action_Object of the BSAO tuple, and the output event will be null. Another remark is to mention that the if-then-else situation in NL sentences is accounted for. Hence, more than one transition may be leaving the same source state in the resulting model.

4. Case study

This section presents the application of our methodology/tool using as case study a space application software product [Santiago et al. 2007]. The test designer starts by the definition of the Dictionary as described in section 2. After that, the scenarios are identified using combinatorial designs. Table 1 shows a possible choice of factors and levels for this case study.

Table 1. Factors and levels for the case study.

Factors	**Levels**						
OpMode	Nom	Init	Safe	Diag	Inv		
Services	Sci	Hk	Dmp	Load	Dg	Tst	Inv
Cmd	TxSci	PrpHk	TxHk	VOpM	LdDat	ExeP	Inv
Storage	MemMg	No	Inv				
HkTime	Def	Min	Max	Inv			

This configuration generates 50 factor combinations in a pairwise design option. These 50 factor combinations shall be interpreted in order to produce 50 scenarios. We will generate test cases related to scenario number 1: {Nom, Sci, TxSci, MemMg, Def}. The test designer then searches in requirements specifications the NL requirements that characterize such scenario.

Table 2. Sample of test cases for scenario 1.

Test Criterion	Test Cases
all-simple-paths	{be power_Power Conditioning Unit/null, be/null, then accomplish_post/null, do not present_irrecoverable problem/null, be/null, ACT-HW-EPP1-OFF/CMD-REC, ACT-HW-EPP2-OFF/CMD-REC}, {be power_Power Conditioning Unit/null, remain/null, wait_60 seconds/null, VER-OP-MODE/INFO-OP-MODE, ACT-HW-EPP1-ON/CMD-REC, ACT-HW-EPP2-ON/CMD-REC, wait_30 seconds/null, wait_600 seconds/null, PREP-HK/CMD-REC, TX-Data-HK/HK-DATA, CH-OP-MODE-NOMINAL/CMD-REC, VER-OP-MODE/INFO-OP-MODE, wait_10 seconds/null, TX-DATA-SCI/SCI-DATA, wait_10 seconds/null, TX-DATA-SCI/SCI-DATA, CH-OP-MODE-SAFETY/CMD-REC, be/null, ACT-HW-EPP1-OFF/CMD-REC, ACT-HW-EPP2-OFF/CMD-REC}, ...

After that, the selected NL requirements are entered via GUI of the tool that supports the methodology, and the test designer asks the tool to generate the Statecharts model. The generated model has 22 BASIC states and 27 transitions. Sample of test cases generated by the GTSC environment using the all-simple-paths test criterion are presented in Table 2. We generated the test suites for the DS, UIO, and all-transitions criteria too. The test designer can then repeat this process for another scenario as described in section 2.

5. Conclusions

This paper presented a methodology with its supporting tool which aims to help test designers to generate test cases based on behavioral models taking into account reactive systems. The methodology assumes that the documents that provide the basis for system and acceptance test case generation, such as requirements specifications, are mostly elaborated using NL. Deriving test cases in an automated manner, as proposed in this paper, starting right from NL documents is a challenge given that NL is ambiguous.

Our methodology showed a formal manner to derive the usage scenarios for system and acceptance test case generation by using combinatorial designs. We believe that our methodology/tool is very easy to apply, but at the same time it is supported by a formal method, not demanding from the user any skills related to formal notations. The automation of the methodology has the potential to decrease the time related to test case generation in complex and real projects.

Future work includes the development of a method that can identify self transitions in the resulting model. Besides, we aim to develop an algorithm to determine hierarchy within the Statecharts created.

References

Cristiá, M., Santiago, V., and Vijaykumar, N. L. (2010). On comparing and complementing two MBT approaches. In *Proceedings...*, Washington, DC, USA. Latin-American Test Workshop (LATW), IEEE Computer Society.

El-Far, I. K. and Whittaker, J. A. (2001). Model-based software testing. In Marciniak, J. J., editor, *Encyclopedia of software engineering*. Wiley, USA.

Fantechi, A. and Spinicci, E. (2005). A content analysis technique for inconsistency detection in software requirements documents. In *Anais...*, pages 245–256. Workshop em Engenharia de Requisitos (WER).

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Mathur, A. P. (2008). *Foundations of software testing*. Dorling Kindersley (India), Pearson Education in South Asia, Delhi, India. 689 p.

Mich, L., Franch, M., and Inverardi, P. (2004). Market research for requirements analysis using linguistic tools. *Requirements Engineering Journal*, 9(1):40–56.

OMG (2007). *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. The Object Management Group (OMG), Needham, MA, USA. 722 p.

Santiago, V., Amaral, A. S. M., Vijaykumar, N. L., Mattiello-Francisco, M. F., Martins, E., and Lopes, O. C. (2006). A practical approach for automated test case generation using Statecharts. In *Proceedings...*, pages 183–188, Los Alamitos, CA, USA. Annual International Computer Software & Applications Conference (COMPSAC) - International Workshop on Testing and Quality Assurance for Component-Based Systems (TQACBS), IEEE Computer Society.

Santiago, V., Mattiello-Francisco, F., Costa, R., Silva, W. P., and Ambrosio, A. M. (2007). QSEE project: an experience in outsourcing software development for space applications. In *Proceedings...*, pages 51–56, Skokie, IL, USA. International Conference on Software Engineering & Knowledge Engineering (SEKE), Knowledge Systems Institute Graduate School.

Santiago, V., Vijaykumar, N. L., Guimaraes, D., Amaral, A. S., and Ferreira, E. (2008). An environment for automated test case generation from Statechart-based and Finite State Machine-based behavioral models. In *Proceedings...*, pages 63–72, Washington, DC, USA. International Conference on Software Testing, Verification and Validation (ICST) - Workshop on Advances in Model Based Testing (A-MOST), IEEE Computer Society.

Sidhu, D. P. and Leung, T. K. (1989). Formal methods for protocol testing: a detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426.

Sleator, D. D. and Temperley, D. (1993). Parsing English with a link grammar. In *Proceedings...*, pages 277–292. International Workshop on Parsing Technologies.

Souza, É. F. (2010). Geração de casos de teste para sistemas da área espacial usando critérios de teste para máquinas de estados finitos. Master Dissertation, 133 p.

Souza, S. R. S. (2000). Validação de especificações de sistemas reativos: definição e análise de critérios de teste. PhD Thesis, 264 p.

Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings...*, pages 173–180. Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology.

Vijaykumar, N. L., Carvalho, S. V., Francês, C. R. L., Abdurahiman, V., and Amaral, A. S. M. (2006). Performance evaluation from Statecharts representation of complex

systems: Markov approach. In *Proceedings...*, pages 183–202, Porto Alegre, RS, Brazil. Congresso da Sociedade Brasileira de Computação (CSBC) - Workshop em Desempenho de Sistemas Computacionais e de Comunicação, Sociedade Brasileira de Computação.