Automating Inspection of Natural Language Requirements and Model-Based Testing

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

[1]Post-Graduate Program in Applied Computing (CAP)

[2]Associated Laboratory of Computing and Applied Mathematics (LAC)

National Institute for Space Research (INPE)
São José dos Campos – SP, BRAZIL

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

Abstract. *Inspection and testing play important roles towards software quality. However, both tasks are usually time-consuming specially if one considers complex projects. Requirements are a valuable starting point for the development of software products, and most of software requirements specifications are still written in natural language. This paper presents a methodology to address the automation of both the analysis of natural language specifications and Model-Based Testing. The goals are to automatically detect problems in natural language requirements, like ambiguity, inconsistency, and incompleteness, and to translate such requirements into behavioral models to support automated testing.*

1. Introduction

Software Quality Assurance involves several activities like planning, measurement, configuration management, walkthroughs, inspection and testing. Inspection is an example of static analysis technique where there is no need to execute the program. It relies on visual examination of deliverables like Software Requirements Specifications (SRSs), Design Documents and source code. Being one the first artifacts produced within the software development lifecycle, a well elaborated SRS usually leads to a smooth development process because designers, programmers, testing teams base their actions on a reliable source of information. However, incomplete, contradictory and ambiguous specifications may cause many mistakes during the development.

Formal methods may be used to express requirements, but they require high expertise and, hence, they are not very common in industrial practice. Unified Modeling Language (UML) use case models are an alternative. However, it is usual that requirements expressed in Natural Language (NL) are the basis for deriving use case specifications in a UML-based software development approach. Furthermore, use case models are often associated with a textual description narrating the behavior through a sequence of actor-system interactions (SINHA et al., 2007). The bottom line is that NL is still the most common approach to express software requirements in practice (MICH et al., 2004). Unfortunately, serious shortcomings exist if a specification is written in NL, making a document unclear, and this impacts on the next artifacts produced within the software development lifecycle.

Test automation is a reality, however, human factor is still very present within a test process. For instance, following a Model-Based Testing (MBT) (EL-FAR; WHITTAKER, 2001) approach, an enviroment may automatically create test cases, but one shall derive behavioral models to be the basis for such enviroment to do its job. Considering system and acceptance testing, where the entire software must be modeled, a test designer should first identify scenarios and develop several models to cover them and, then a Model-based tool can be used for test case generation. These manual activities accomplished by a test designer are usually time-consuming.

This paper presents a methodology to address the automation of both the analysis (a particular type of inspection) of NL specifications and MBT. The goals are to automatically detect defects in NL requirements, like ambiguity, inconsistency, and incompleteness, and to translate such requirements into behavioral models to support the automation of system and acceptance testing. Description Logics will be used to support both activities (NARDI; BRACHMAN, 2003).

2. Natural Language Specifications

According to a recent survey, 95% of the requirements documents found in industry are written in common (79%) or structured (16%) natural language (MICH et al., 2004). There is a lack of methodologies and tools for NL requirements analysis. This section presents some approaches addressing this issue, such as the *Quality Analyzer for Requirements Specification* (QuARS) tool (GNESI et al., 2005). QuARS was developed based on a quality model for the expressiveness property (mainly ambiguity and poor readability). QuARS seems to be a scalable tool, however, the analysis it performs is limited to syntax-related issues of NL requirements documents addressing ambiguity. Besides, the tool does not perform true automation detection of inconsistency and incompleteness.

Another environment that supports modeling and analysis of NL requirements is CIRCE (AMBRIOLA; GERVASI, 2006). CIRCE uses a domain-based parser called CICO to parse and transform NL requirements into a forest of parse trees. A requirements specification is considered as a set of *designations*, a set of *definitions*, and a set of *requirements*. CIRCE assumes that the requirements are expressed in restricted NL: there are formal rules which define the controlled language accepted. Besides, the domain must be defined by a user by means of designations and definitions written using a formal syntax. The tool is interesting but expressing the domain may be difficult, tiresome and time-consuming because it is necessary to declare designations, using lots of tags, and definitions requiring from a requirements Engineer to perform a deep analysis of the NL requirements. Furthermore, it is not completely evident that the tool can properly deal automatically with ambiguity and inconsistency, scalability is an issue and it is very likely that a user needs to write formal rules, which drive the CICO's parsing algorithm, when using the tool.

The *Natural Language – Object Oriented Production System* (NL-OOPS) tool supports analysis of unrestricted NL requirements by extracting classes and their associations for use in creating class models (MICH, 1996). The unrestricted NL analysis is obtained using as a core the NL processing system *Large-scale, Object-based, Linguistic Interactor, Translator, and Analyser* (LOLITA). LOLITA is built around a large graph called *SemNet*, a particular form of conceptual graph, which holds knowledge that can be accessed, modified or expanded using NL input. NL-OOPS allows detection of ambiguities, but probably not all possible types, but there is no evidence that it supports automated detection of incompleteness and inconsistency. Moreover, the tool is not scalable.

Gervasi and Zowghi proposed a formal framework for identifying, analyzing, and managing inconsistency in requirements derived from multiple stakeholders and expressed in controlled NL (GERVASI; ZOWGHI, 2005). A prototype tool, CARL, was developed incorporating all the techniques described in the paper. Requirements expressed in controlled NL are first automatically parsed and translated into propositional logic formulae. Once the specification is represented as sets of propositional logic formulae, a theorem prover and a model checker are used aiming at detecting inconsistencies. Limitations of CARL include no support for automated detection of incompleteness and ambiguity, scalability and, like CIRCE (AMBRIOLA; GERVASI, 2006), there is the same problem regarding the likely need to write new formal rules depending on the domain.

3. Translations of Notations and Model-Based Testing

The translation of specifications elaborated in one notation to another one may be beneficial for software testing purposes. Approaches that translate simpler notations, like NL or UML diagrams, to formal methods can be quite convenient because they relieve testing professionals from the cost of using a formal method but, at the same time, provide the requirements converted to a formal way for verification and testing. Some work regarding this topic follows.

The work of Gervasi and Zowghi is an example of transformation of NL requirements into formal method (GERVASI; ZOWGHI, 2005). Sinha et al. (SINHA et al., 2007) demonstrated how a combination of UML use case and class diagrams can be converted to an Extended Finite State Machine (EFSM). Fröhlich and Link presented a system testing method based on textual descriptions of UML use cases (FRÖHLICH; LINK, 2000). They translated a use case description into a UML state machine and, after that, they applied Artificial Intelligence planning techniques to derive test suites satisfying the coverage testing criterion which asserts that all transitions of the UML state machine must be traversed at least once.

Sarma and Mall proposed a system testing approach to cover *elementary transition paths* (SARMA; MALL, 2009). The technique relies on the derivation of a System State Graph (SSG) based on UML 2.0 use case, sequence and Statecharts diagrams. The test criterion which their method aims to satisfy is *transition path coverage* where each elementary transition path *p* of the SSG must be exercised at least once by a test suite *T*. One major limitation of their approach is not considering *loops* in sequence diagrams, given that a loop is either not executed at all or it is executed only once. Thus, the authors did not address one of the major problems in path testing because, in general, a program containing loops will have an infinite or undetermined number of paths.

The testing community tends to consider MBT as a type of testing in which tests are derived from software behavioral models (EL-FAR; WHITTAKER, 2001). This definition includes formal methods specifications and other notations, like UML models. Finite State Machines (FSMs) (LEE; YANNAKAKIS, 1996) and Statecharts (HAREL, 1987) are a few examples of modeling techniques commonly used for testing. Once a system is modeled as a state-transition diagram representing an FSM, several methods like Transition Tour (TT), Distinguishing Sequence (DS), Unique Input/Output (UIO), W (SIDHU; LEUNG, 1989), switch cover (PIMONT; RAULT, 1976) and state counting (PETRENKO; YEVTUSHENKO, 2005) can be used to generate test cases.

Several approaches have been proposed to generate test cases from Statecharts models. Binder (BINDER, 1999) adapted the W method to a UML context and named it round-trip path testing. Souza proposed a family of testing coverage criteria, the *Statechart Coverage Criteria Family* (SCCF), for models in Statecharts (SOUZA, 2000). Test requirements established by the SCCF criteria are obtained from the Statecharts reachability tree. Briand et al. (BRIAND et al., 2004) showed a simulation and a procedure to analyze cost-efficiency of three criteria proposed by Offutt and Abdurazik (OFFUTT; ABDURAZIK, 1999) and the very same round-trip path.

The *Geração Automática de Casos de Teste Baseada em Statecharts* (GTSC) is an environment that allows test designers to model software behavior using Statecharts and/or FSMs in order to generate test cases automatically based on some test criteria (methods) for FSM and some for Statecharts (SANTIAGO et al., 2008a). At present, GTSC has implemented switch cover, UIO and DS test criteria for FSM models and two test criteria from SCCF, all-transitions and all-simple-paths, targeting Statecharts models. GTSC has been successfully used for model-based test case generation regarding software products embedded into experiment on-board computers of scientific satellites under development at CEA/INPE (SANTIAGO et al., 2008a).

4. The Methodology

This section presents the proposed methodology, shown in Figure 1, which will try to encompass static and dynamic techniques. The automated analysis of NL requirements is the static part of the methodology. The bold rectangles in Figure 1 indicate tools that will be developed to build an NL Processing System. The only tool that will not be developed is the *Part-Of-Speech (POS) Tagger* and the first option is to use the Stanford POS Tagger (TOUTANOVA et al., 2003). The main role of the tagger is to identify nouns, verbs, adjectives to support the Knowledge Base (KB) inference of the system.

The KB of the system will be encoded using Description Logics (DL), a formalism for representing knowledge, as well as some important basic notions underlying all systems that have been created in the DL tradition (NARDI; BRACHMAN, 2003). A DL KB is typically composed of a TBox and an ABox. The TBox contains intensional knowledge in the form of a terminology and is built by means of declarations that describe general properties of concepts. The ABox contains extensional knowledge that is specific to the individuals of the domain of discourse.

The methodology requires that a user provides a lightweight ontology using a Graphical User Interface (GUI). Indeed, the user will perform a mapping among domain words and concepts, sets or classes of individual objects, in the ontology. An ontology, like the one shown in Figure 2, will exist regarding the domain of software embedded into satellite on-board computers. Based on reasoning services provided by DL, like subsumption (NARDI; BRACHMAN, 2003), the KB (TBox and ABox) can be improved to deal with, for instance, the problem of incompleteness specifications. The POS Tagger will provide syntactic category in order to help in the KB inference process, given that it is possible to identify concepts (nouns, domain entities) and actions (verbs), the latter possibly characterizing relationships (roles). In Figure 2, an example of concept is *OnBoardComp* and a role is ∃*hasType.Message*. The lightweight term implies that it will not be required to use any type of formalism to provide such mapping. Besides, the idea is that a user does not take a long time to do that, because the design ontology will be defined within the tool that supports the methodology.

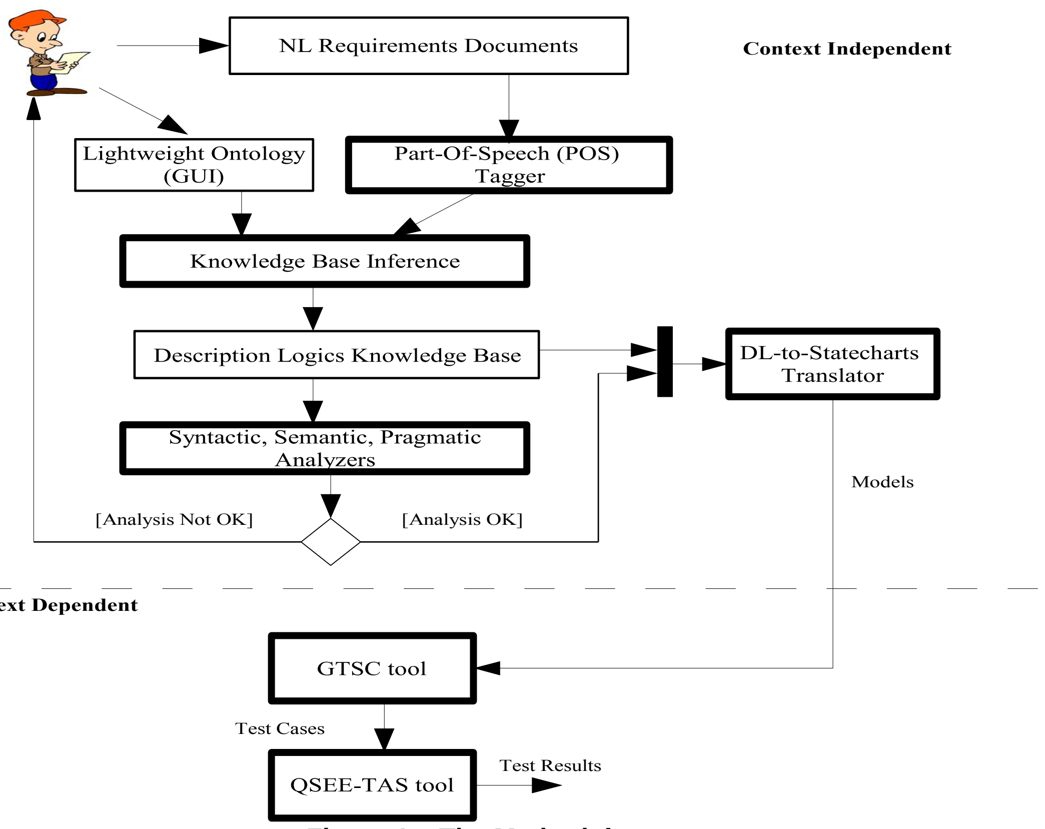
NL Requirements Documents
Context Independent
Lightweight Ontology (GUI)
Part-Of-Speech (POS) Tagger
Knowledge Base Inference
Description Logics Knowledge Base
DL-to-Statecharts Translator
Syntactic, Semantic, Pragmatic Analyzers
Models
[Analysis Not OK]
[Analysis OK]
Context Dependent
GTSC tool
Test Cases
QSEE-TAS tool
Test Results

Figure 1 – The Methodology.

1. OnBoardComp ⊑ SatelliteCompSubsytem
2. OnBoardSoft ⊑ OnBoardComputer
3. CommProtocol ⊑ ∃hasImplementation.OnBoardSoft ⊓ ≥2hasComputer
4. Command ⊑ CommProtocol
5. Response ⊑ CommProtocol
6. Message ≡ Command ⊔ Response
7. Field ⊑ (∃hasType.Message ⊓ ∃hasErrorDetection.Message) ⊓ ≥2hasField

Figure 2 – A piece of ontology (TBox) for space embedded software.

Having the DL KB inferred, semantic and pragmatics analysis may be performed aiming at detecting ambiguity, inconsistency and incompleteness at the formal level. These defects will be informed to the user so that he/she can take the proper actions such as rewriting the requirenments and/or improving the ontology mapping.

Once the requirements Engineer decides the NL specification is adequate, the dynamic feature of the methodology can take place. This feature refers to system and acceptance automated test case generation by the translation of the DL KB into Statecharts models. Recall that such a kind of testing demands the modeling of the entire system. The natural way to address this activity is the divide and conquer strategy, where a test designer breaks down the system based on functional and possibly non-functional requirements. This strategy tries to diminish the impact of the existing state explosion problem related to MBT. Models are then derived to address each functionality and, in this way, it is more feasible to generate test cases based on such models. Consider the following three requirements adapted from the specification of SWPDC, a software product developed under the scope of the *Qualidade do Software Embarcado em Aplicações Espaciais* (QSEE) research project (SANTIAGO et al., 2007):

[ET003] On the beginning of its operation, SWPDC will accomplish a POST (Initiation Operation Mode) to determine whether the Payload Data Handling Computer (PDC) is healthy and adequate to operate. If any unrecoverable problem was detected within PDC, this computer will remain in the Initiation Operation Mode and this problem will not be propagated to the On-Board Data Handling Computer (OBDH).

[ET012] In the case that none unrecoverable problem was detected within PDC, after the initiation process, SWPDC shall automatically change the PDC Operation Mode to Safety.

[ET013] The PDC/SWDPC will be available to communicate with the OBDH only after 1 minute has elapsed since the initiation process.

Such requirements define a small part of a particular scenario that a test designer must identify in order to elaborate the behavioral model. Figure 3 shows a simple ontology (TBox) related to the transformation of DL formalism into a reactive system Statecharts model. Figure 4 shows the ABox considering the tree requirements abovementioned.

The basic idea to transform such Knowledge Base into a Statecharts model is considering left-hand side concepts of TBox as *states* and right-hand side roles as input or output *events* within transitions in the model. Note that the roles has IN and OUT terms meaning the input and output events within transitions, respectively. Also note that there are roles that are type *ev* and others that are type *evcond*. These terms model the situations where there is just an event without any guard condition (ev) and event with guard condition (evcond). Besides, a prefix *no* implies that such type of event (e.g. *no-evcond* means that no event with condition occurs to fire the transition) does not occur. The *DL-to-Statecharts Translator* component (Figure 1) will then examine the ABox and, following its structure, it can generate the Statecharts model.

In order to clarify how the model can be generated, see statements 2, 3 and 4 in Figure 4. In 2, the next (destination) state will be *Initiation Mode*. In 3, the input event without guard condition is *switchPDCOn* and the source state is *PDCOff* (which is also the initial state of the model; see statement 1). In 4, the output event is *start60s*, given that requirement [ET013] specifies a one-minute delay in order SWPDC/PDC can be able to communicate with the OBDH. Figure 5 shows the Statecharts model regarding the scenario described by such requirements.

Obviously, the derivation of the behavioral model requires a precise KB (TBox and ABox), and this shows how important is the role of the *Knowledge Base Inference* component within this methodology. Moreover, it is important to note that there should exist different KBs within the system, like the KB resulted from the analysis of NL requirements and the KB regarding the translation from DL to behavioral models to support testing.

1. InitialState ≡ ∃hasINITIALSTATE.System
2. NextState ⊑ (((∃hasIN_ev.InitialState ⊓ ∃hasIN_no-evcond.InitialState) ⊔ (∃hasIN_no-ev.InitialState ⊓ ∃hasIN_evcond.InitialState)) ⊓ (∃hasOUT_event.InitialState ⊔ ∃hasOUT_null-out.InitialState)) ⊔ (((∃hasIN_ev.OtherState ⊓ ∃hasIN_no-evcond.OtherState) ⊔ (∃hasIN_no-ev.OtherState ⊓ ∃hasIN_evcond.OtherState)) ⊓ (∃hasOUT_event.OtherState ⊔ ∃hasOUT_null-out.OtherState))
3. NoMovement ⊑ ∃hasIN_undef-event.InitialState ⊔ ∃hasIN_undef-event.OtherState

Figure 3 – An example of a ontology (TBox) regarding the DL to Statecharts translation.

1. InitialState(PDCOff)
2. NextState(InitiationMode)
3. hasIN_ev(PDCOff, switchPDCOn)
4. hasOUT_event(PDCOff, start60s)
5. NextState(SafetyMode)
6. hasIN_evcond(InitiationMode, end60s [POSTStatusOk])
7. hasOUT_event(InitiationMode, changeToSafety)
8. NextState(InitiationMode)
9. hasIN_evcond(InitiationMode, noend60s [⌐POSTStatusOk])
10. hasOUT_null-out(InitiationMode, null)
11. NextState(InitiationMode)
12. hasIN_evcond(InitiationMode, end60s [⌐POSTStatusOk])
13. hasOUT_null-out(InitiationMode, null)

Figure 4 – An example of a ABox addressing three requirements of SWPDC.

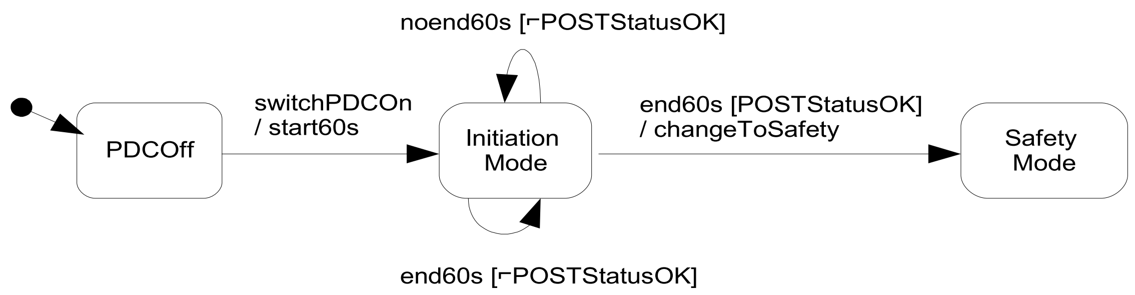

Figure 5 – The derived Statecharts model.

Once the Statecharts models are derived, the GTSC environment may be used for test case generation and the *QSEE-Teste Automatizado de Software* (QSEE-TAS) tool may automatically execute the test cases (SANTIAGO et al., 2008b).

5. Conclusions

This paper presented a review literature regarding NL requirements and MBT, and also a methodology to address the analysis of NL specifications aiming to identify automatically defects such as ambiguity, inconsistency, and incompleteness. The methodolgy also intends to translate such requirements, actually their Description Logics representantion, into Statecharts behavioral models to support the automation of system and acceptance testing. Preliminary developments related to this work are the ontologies designed to address space embedded software and DL to Statecharts translation, and a mechanism to derive behavioral models for testing purposes based on an adequate ABox.

Future work will include improving the proposed ontologies, the development of the tools shown in Figure 1 and the POS Tagger integration with such tools.

References

AMBRIOLA, V.; GERVASI, V. On the Systematic Analysis of Natural Language Requirements with CIRCE. Automated Software Engineering, v. 13, n. 1, p. 107-167, 2006.

BINDER, R. V. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional, 1999. p. 1248.

BRIAND, L. C.; LABICHE, Y.; WANG, Y. Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statechart. In: 26th Int. Conf. on Software Engineering, 2004, Edinburgh, Scotland, UK. p. 86-95.

EL-FAR, I. K.; WHITTAKER, J. A. Model-Based Software Testing. In: MARCINIAK, J. J. (ed.). Encyclopedia of Software Engineering. Wiley, 2001. p. 1584.

FRÖHLICH, P.; LINK, J. Automated Test Case Generation from Dynamic Models. LNCS, v. 1850, p. 472-491, 2000.

GERVASI, V.; ZOWGHI, D. Reasoning about Inconsistencies in Natural Language Requirements. ACM Transactions on Software Engineering and Methodology, v. 14, n. 3, p. 277-330, 2005.

GNESI, S.; LAMI, G.; TRENTANNI, G. An automatic tool for the analysis of natural language requirements. International Journal of Computer Systems Science and Engineering, v. 20, n. 1, p. 1-13, 2005.

HAREL, D. Statecharts: A visual formalism for complex systems. Science of Computer Programming, v. 8, p. 231-274, 1987.

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines – A Survey. Proceedings of the IEEE, v. 84, n. 8, p. 1090-1123, 1996.

MICH, L. NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. Natural Language Engineering, v. 2, n. 2, p. 161-187, 1996.

MICH, L.; FRANCH, M.; INVERARDI, P. Market research for requirements analysis using linguistic tools. Requirements Engineering Journal, v. 9, n. 1, p. 40-56, 2004.

NARDI, D.; BRACHMAN, R. J. An Introduction to Description Logics. In: BAADER, F.; CALVANESE, D.; MCGUINNESS, D.; NARDI, D.; PATEL-SCHNEIDER, P. (eds.). The Description Logic Handbook. Cambrige University Press, 2003. p. 505.

OFFUTT, J.; ABDURAZIK, A. Generating Tests from UML Specifications. LNCS, v. 1723, p. 416-429, 1999.

PETRENKO, A.; YEVTUSHENKO, N. Testing from Partial Deterministic FSM Specifications. IEEE Transactions on Computers, v. 54, n. 9, p. 1154-1165, 2005.

PIMONT, S., RAULT, J. C. A Software Reliability Assessment Based on a Structural and Behavioral Analysis of Programs. In: 2nd Int. Conf. on Software Engineering, 1976, San Francisco, CA, USA. p. 486-491.

SANTIAGO, V.; MATTIELLO-FRANCISCO, F.; COSTA, R.; SILVA, W. P.; AMBROSIO, A. M. QSEE Project: An Experience in Outsourcing Software Development for Space Applications. In: The 19th Int. Conf. on Software Engineering & Knowledge Engineering, 2007, Boston, USA. p. 51-56.

SANTIAGO, V.; VIJAYKUMAR, N. L.; GUIMARÃES, D.; AMARAL, A. S.; FERREIRA, E. An Environment for Automated Test Case Generation from Statechart-based and Finite State Machine-based Behavioral Models. In: 4th A-MOST, 1st IEEE Int. Conf. on Software Testing Verification and Validation, 2008, Lillehammer, Norway. p. 63-72.

SANTIAGO, V.; SILVA, W. P.; VIJAYKUMAR, N. L. Shortening Test Case Execution Time for Embedded Software. In: 2nd IEEE Int. Conf. on Secure System Integration and Reliability Improvement, 2008, Yokohama, Japan. p. 81-88.

SARMA, M.; MALL, R. Automatic generation of test specifications for coverage of system state transitions. Information and Software Technology, v. 51, n. 2, p. 418-432, 2009.

SIDHU, D. P.; LEUNG, T. K. Formal Methods for Protocol Testing: A Detailed Study. IEEE Transactions on Software Engineering, v. 15, n. 4, p. 413-426, 1989.

SINHA, A.; PARADKAR, A.; WILLIAMS, C. On Generating EFSM models from Use Cases. In: 6th Int. Workshop on Scenarios and State Machines, 29th Int. Conf. on Software Engineering, 2007, Minneapolis, MN, USA. p. 1-8.

SOUZA, S. R. S. Validação de Especificações de Sistemas Reativos: Definição e Análise de Critérios de Teste. f. 264. PhD Thesis, Universidade de São Paulo, São Carlos, SP, Brazil, 2000.

TOUTANOVA, K.; KLEIN, D.; MANNING, C. D.; SINGER, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In: Conf. of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, 2003, Edmonton, Canada. p. 173-180.