Requirements Verification

Legal Challenges in Compliance Testing

Jean-Pierre Corriveau and Vojislav Radonjic
School of Computer Science
Carleton University
Ottawa, Canada
jeanpier@scs.carleton.ca

Wei Shi
Faculty of Business and I.T.
University of Ontario, Institute of Technology
Oshawa, Canada
Wei.Shi@uoit.ca

***Abstract*—Compliance is generally understood as the documenting and auditing of evidence deemed sufficient to demonstrate conformance to a rule, a specification, a policy or a law. In this paper, we consider, in the specific context of software development, what are the legal and technical challenges raised by such an understanding of compliance. More specifically, we ask a) what is the nature of this evidence; b) how can sufficiency be defined, and c) how precisely defined is the task of auditing this evidence.**

Keywords—compliance; traceability; compliance testing; test specifications; executable tests

I. Introduction

Zave [1] offers the following definition for Requirements Engineering (hereafter RE):

"*Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families*."

In commenting on this definition, Nuseibeh and Easterbrook [2] remark:

"*This definition is attractive for a number of reasons. First, it highlights the importance of "real-world goals" that motivate the development of a software system. These represent the 'why' as well as the 'what' of a system. Second, it refers to "precise specifications". These provide the basis for analyzing requirements, validating that they are indeed what stakeholders want, defining what designers have to build, and verifying that they have done so correctly upon delivery*."

It is this last task, namely the verification that an actual implementation complies with the requirements of stakeholders that is the focus on this paper.

Lin and Yu [3] observe that both goal-driven and scenario-based approaches have proven useful in the context of RE: the former for the capturing "why" the data and functions are there, and whether they are sufficient for achieving the high-level objectives whereas scenarios present possible ways to use a system to accomplish some desired functions or implicit purposes. The User Requirements Notation [4] indeed offers an international standard combining a goal-oriented requirements language and a scenario-based notation (namely use case maps [5]). An overview of this conceptual and methodological framework is provided in [6].

As is, the URN offers limited test case generation [4, 7], and the executability of such test cases is generally not addressed. In fact, this is typical of most model-based approaches to testing [8, 9, 10]. Let us briefly elaborate. While many tools claim they generate executable tests, they generally require that complex adapters be written in order to 'connect' the generated tests to the system under test (e.g., [11] with respect to TTCN-3 executability). This manual task is not only complex [*Ibid.*] and time-consuming but potentially error-prone. Also, this so-called "glue code" is implementation-specific and thus, both its reusability across systems under test and its maintainability are problematic.

Unfortunately, we believe that from a legal standpoint, these two limitations (namely: limited test generation and serious issues with respect to the executability of the generated tests) point to several serious challenges for requirements verification. It is this contention that we elaborate upon in the rest of this paper. More specifically, compliance is generally understood as the documenting and auditing of evidence deemed sufficient to demonstrate conformance to a rule, a specification, a policy or a law. From a legal viewpoint, such a definition raises three important questions: a) what is the nature of this evidence; b) how can sufficiency be defined, and c) how precisely defined is the task of auditing this evidence? Those are the questions that we consider below in the specific context of requirements engineering for the development of a software system.

II. Automated Requirements Verification

A. Software Compliance Testing

Regardless of which model-centric or code-centric development process is adopted, industrial software production ultimately and necessarily requires the delivery of an executable implementation. Furthermore, it is generally agreed that the quality of this implementation is of the utmost importance, as reflected by the recent CISQ initiative [12].

978-1-4799-2030-3 /14/$31.00 ©2014 IEEE

Consequently, the requirements of stakeholder(s) must be verified against the actual behavior of the implementation under test (IUT), a task that is typically called "conformance" or equivalently "compliance" testing.

For example, in the context of software development outsourcing [13, 14, 15], a *contract* is required in order to define a) what services are requested from the contracted entity and b) how these services are to be delivered to the satisfaction of the contractor. Because outsourcing is often motivated by cost reduction considerations, potential legal battles are obviously to be avoided. This is particularly important in the context of offshore outsourcing where internationalization can quickly and dramatically complicate such battles. Consequently, a software offshore outsourcing contract will clearly gain in including, among its quality assurance facets, the specification of a systematic objective approach to the conformance testing of the functional and non-functional requirements of the system to be delivered (hereafter STBD).

The key postulate of this paper is that the above observation is in fact relevant to any software development endeavor. That is:

1) We postulate that compliance testing is a critical aspect of any software development endeavor: without it there is no quality control, which is an untenable position from both a business and a legal viewpoint.

2) We also postulate that a legal perspective regarding compliance testing demands a systematic and objective approach for this task. That is, in a legal dispute on whether or not a software system satisfies the requirements of its stakeholder(s): a) a contract will be required and b) this contract will have to facilitate and optimize the objective assessment of its completion to the satisfaction of the stakeholder(s). Without a contract, there is no case, and without a systematic and objective method for the evaluation of this contract, establishing facts (as opposed to opinions) is greatly jeopardized.

It is in light of these two postulates that we now return to the three questions raised at the end of the previous section. Each of these is addressed separately in the next three subsections.

B. Evidence of Compliance

What constitutes evidence of compliance in the context of software development? As previously mentioned, requirements engineering generally relies on goal-driven and scenario-based models (separately or in combination [3]). It is important to acknowledge that such models are necessary as they are meant to act (implicitly or explicitly) as oracles [16], that is, because they should define what is expected of the STBD. But they do not provide evidence per se. Proceeding from the very definition of compliance testing [8, 16], evidence of compliance of a software system to its requirements must consist in comparing actual functional and non-functional behavior to expected functional and non-functional behavior. This entails: a) the specification of expected behavior, b) the specification of the set of actual tests executed on the IUT, c) the execution of these tests and their comparison to the corresponding expected behavior. In turn, these three tasks lead to three challenges namely: a) testability, b) traceability and c) executability.

First, the testability of a model is defined as its ability to have tests generated in a systematic (read algorithmic) way from it [16]. Goal models not only address what tasks need to be accomplished by STBD but also the causal relationships that must exist between the goals of the stakeholders/customers and such tasks. While such models are useful, especially with respect to elicitation and analysis of non-functional requirements, they are not testable per se. Consequently, a legal demonstration of compliance with respect to non-functional requirements is quite problematic. Conversely, scenario models correspond to more precise functional specifications of the STBD and thus generally lend themselves to some form of test generation (in the form of path traversal, as explained in [*Ibid.*]). However, it is important to understand that such generated tests are not readily executable against an IUT. Instead, they act as specifications for what needs to be tested: it is left to a programmer to code the corresponding test cases, execute them and compare them to expected behavior and report the outcome of such comparisons. Alternatively, as previously mentioned, a programmer may have to develop complex adapters (e.g., for TTCN-3 [11] or tools such as Spec Explorer [17]) that enable the generated tests to be ‘executed’ against the STBD. We use quotes around the word ‘executed’ to emphasize ‘execution’ here is typically tool-specific and limited. For example, TTCN-3 tools require the use of ‘ports’ for the specification and the implementation to ‘communicate’. Similarly, Spec Explorer ‘reduces’ testing to the matching of parameter and return values of public procedure calls.

Legally, such an approach opens the door to questioning the correspondence (or traceability) that should exist between the requirements of the stakeholders, the tests generated from scenario models, and the actual tests run against an IUT. Moreover, whereas testability is an issue of feasibility (i.e., there is a tool that generates tests or there is not), traceability is possibly significantly more complex to establish (especially in the presence of notation specific concepts such as TTCN-3 ports and tool-specific adapters). The reason for this observation is simple: there is no one-to-one relationship between requirements and tests (be they generated or actual). In fact, each requirement typically is associated (manually or through some requirement tracking tool) with several paths across several scenarios. And, through the technique of path sensitization [16], each such path can be associated with several generated tests. To further complicate matters, in practice, the use of boundary value analysis [*Ibid.*] will lead a programmer to code up several actual tests for each generated tests. Thus, in summary, (ideally automated) support for this two-tiered model of traceability is necessary if one is to argue that the actual tests used for compliance testing do correspond to the requirements of the stakeholder(s) (via links to generated tests). But it must be emphasized that even with such support for traceability, there is still room for litigation: the existence of links between requirements, generated tests and actual tests by no means guarantees that these links are semantically correct. This is especially an issue when actual tests are not obtained in an automated way from the tests generated from the scenario models: a traceability link between a generated test and an actual test does not semantically guarantee that the actual test addresses the generated one. The link merely captures the belief of the programmer (who coded the actual tests) that the actual test does address the generated test to which it is linked.

Finally, with respect to the meaning of 'executability', it is important to remark that several model-based tools claim to offer test generation and test execution. But in fact such 'test execution' often consists in *symbolic execution*, that is, is carried out using a (typically state-based) model of the IUT, not the actual IUT [17]. From a legal standpoint, the relevance of such simulations to litigation on compliance is quite dubious (for both functional and non-functional requirements). Indeed, the software industry has a long history of projects (e.g., in telephony, in reservation systems, in e-health, etc.) whose catastrophic actual performance at the very start of their usage led to their quick shutdown despite favorable simulations using highly sophisticated mathematical models (such as layered queue networks). In other words, simulations, regardless of their pervasiveness in the literature, are just that, *simulations*. They may provide indicators of what to expect of an actual system. But they cannot guarantee that they will correspond to actual executions; they do not provide evidence of compliance, for the object of litigation at hand is the compliance of an IUT (that is, of an actual running system) against the requirements of the stakeholders.

C. Sufficient Compliance

Beyond the challenges stemming from testability, traceability and executability, unavoidably the issue of sufficiency will be at the heart of software compliance testing. The question is simple: how much should the IUT be tested in order to demonstrate that it complies to its requirements? The difficulty is that it is widely accepted that, except for trivial systems, software testing is generally incomplete [16]. Put another way, for most industrial software systems, there is typically an intractable number of situations (e.g., combinations of paths through scenario models) to test. (Additional technical problems such as the combinatorial explosion of tests required for the testing of complex Boolean expressions [16, chapter 6], in fact compound this problem.) Consequently, it is unrealistic to demand that all possible paths from all scenarios be selected, instantiated, tested and matched. Instead, the contractor and contracted must agree and document in the compliance contract how much coverage will be required. Let us elaborate.

Coverage will have to address how extensive the suite of actual tests is to be. This requires the contractor and the contracted to agree on a) a set of scenarios, b) a set of paths through these scenarios, and c) for each path, a set of path instantiations and corresponding actual test cases to run. Several challenges stem from such a task: The two parties must agree on a scenario model notation that both can use to discuss the behavior of the STBD. Then they must agree on how complete the set of scenarios is to be, which assumes tracing back scenarios to requirements in order to ensure each requirement is addressed. Then tests (in the form of paths through scenarios) must be generated. Ideally this should be automated, otherwise traceability is in jeopardy and such an unsystematic approach may lead to critical omissions. That is, a tool that generates a set of paths to test from a scenario model has the advantage of being systematic and reusable. In this case, the two parties must agree that the generation algorithm of the tool does generate adequate coverage of the requirements. This can be somewhat legally problematic inasmuch as it requires a (typically non-technical) stakeholder to agree to the use of a test generation tool whose algorithm is unlikely to be fully understood by this stakeholder. Similarly, once paths have been selected for testing, specific path instantiations must be chosen. Again a stakeholder is asked to agree to a specific set of tests without necessarily understanding the techniques (such as equivalence partitioning and boundary value analysis [16]) that led to the generation of this set. In other words, while the use of algorithms and tools to automate the generation/selection of a suite of actual tests brings systematicity to agreeing on sufficiency, it also 'forces' the contractor in relying on technical issues s/he may not fully master. The idea of 'sneak paths' illustrates this point. Binder [*Ibid.*] explains that, beyond test generation techniques, a test suite should always include additional ("sneak") paths corresponding to 'tricky' situations envisioned by the tester but missed by the generation tool. A contractor versed in the systematic way of thinking required by scenario testing may come up with such 'sneak paths' (which should naturally be added to the test suite agreed upon). But, in practice, it is likely to be the tester of an IUT who may have such intuitions and suggest such additions. And thus, ultimately, the contractor who provides initial (typically fairly abstract) requirements is asked to agree to a detailed test suite that he must believe to be sufficient with respect to these requirements. This dilemma is further complicated if compliance is not restricted to a static interpretation but, instead is seen as diachronic. In fact, the difficulties resulting from having compliance possibly evolve over time are such that, legally, this must simply be ruled out.

D. Compliance Testing

Beyond the challenges stemming from testability, traceability, executability and coverage, we identify two other sources for possible legal disputes with respect to software compliance testing.

First, we must consider the process of compliance testing per se, that is, how tests are run and evaluated. Without going into technical details (see [14, 15, 18]), we remark that monitoring the execution of path in an IUT is not a trivial exercise. Nor is deciding whether this execution matches or not the expected behavior. Any ad hoc approach to these tasks is error-prone and thus should be avoided. Instead, the compliance contract should require that tests be automatically instrumented, monitored and evaluated [18] in order to guarantee systematicity and objectivity. The alternative is legally unacceptable: the IUT tester would be in fact judge and party since this tester would specify the tests, run them, and evaluate them.

Second, we observe that in many industrial software development projects, compliance is not defined in terms of a test suite but rather in terms of a percentage of successful tests over a certain percentage of this test suite! For example, a release will occur when 90% of the test suite has been exercised with a success rate of 98% (where 'success' means the actual behavior matches the expected one). As this is quantitative information, it does not represent a legal challenge to verify. It is therefore left to the two parties to decide whether or not to refer to such percentages. In our experience, such considerations are business-driven and ideally left out of a compliance contract because: a) such percentages are meaningless without prioritization: some tests are critical and must work and b) introducing a prioritization scheme over tests entails creating another source of potential dispute between the two parties.

III. Conclusion

In this paper we have identified several facets of software compliance testing that 'open the door' to litigation. To do so, we have assumed requirements would be captured in state-of-the-art goal-driven and scenario-based models. In fact, when considering the paucity (with respect to testability, traceability and executability) of the models still used for defining the detailed requirements of multi-million systems (e.g., [19]), we must conclude that the current situation is much worse than what we have described.

Acknowledgments

Support from the Natural Sciences and Engineering Council of Canada is gratefully acknowledged.

References

[1] P. Zave, “Classification of research efforts in requirements engineering,” ACM Computing Surveys, 29(4), pp.315-321, 1997.

[2] B. Nuseibeh and S. Easterbrook, “Requirements engineering: a roadmap,” Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, pp.35-46, June 2000.

[3] L. Liu and E. Yu, “From requirements to architectural design - using goals and scenarios,” ICSE-2001 Workshop: From Software Requirements to Architectures (STRAW 2001), Toronto, Canada. pp.22-30, May 2001.

[4] D. Amyot and G. Mussbacher, “User requirements notation: the first ten years, the next ten years,” Journal of Software, Vol. 6(5), pp.747-768, May 2011.

[5] R. Buhr and R. Casselman, Use Case Maps for Object Oriented Systems, Prentice-Hall, USA, 1995.

[6] D. Amyot, “Introduction to the user requirements notation: learning by example,” Computer Networks: The International Journal of Computer and Telecommunications Networking, Elsevier Inc., New York, vol. 42(3), pp.285-301, June 2003.

[7] D. Amyot, M. Weiss and L. Logrippo, “Generation of test purposes from Use Case Maps,” Journal of Computer Networks, vol. 49(5), pp.643-660, 2005.

[8] A. Bertolino, “Software testing research: achievements, challenges and dreams,” Future of Software Engineering (FOSE '07), IEEE Press, Minneapolis, pp.85-103, May 2007.

[9] J.-P. Corriveau, and W. Shi, “Traceability in acceptance testing,” Journal of Software Engineering and Applications, Vol. 6(10A), pp.36-46, 2013.

[10] M. Shafique and Y. Labiche, “A systematic review of model based testing tool support,” Technical Report SCE-10-04, Carleton University, 2010.

[11] B. Stepien and L. Peyton, “A comparison between TTCN3 and Python,” TTCN-3 User Conference, Madrid, Spain, June 2008.
http://www.site.uottawa.ca/~bernard/A%20comparison%20between%20ttcn-3%20and%20python%20v%2012.pdf

[12] M. Surhone, M. Tennoe and S. Henssonow, CISQ, Betascript Publishing, 2010.

[13] B. Meyer, “The unspoken revolution in software engineering,” IEEE Computer, 39(1), pp.121–124, 2006.

[14] J.-P. Corriveau, “Testable requirements for offshore outsourcing,” Software Enginnering Approaches for Offshore and Outscourced Development (SEAFOOD), Zurich, Switzerland, February 2007 (LNCS 4716, pp.34-43).

[15] D. Arnold, J.-P. Corriveau and W. Shi, “Reconciling offshore outsourcing with model-based testing,” Software Engineering Approaches for Offshore and Outscourced Development (SEAFOOD 2010), Peterhorf (Saint Petersburg), Russia, pp.6-22, June 2010.

[16] R. Binder, Object-Oriented Testing, Addison-Wesley, New York, 2000.

[17] “Spec Explorer Visual Studio Power Tool,” http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745

[18] J.-P. Corriveau and W. Shi, “Generating verifiable contracts,” Software Engineering Research and Practice, Las Vegas, pp.315-321, July 2011.

[19] Software Communication Architecture draft spec.,
http://www.public.navy.mil/jpeojtrs/sca/Pages/default.aspx