Formal Methods in Industrial Software Standards Enforcement

Alexey Grinevich, Alexey Khoroshilov, Victor Kuliamin, Denis Markovtsev, Alexander Petrenko, and Vladimir Rubanov

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
{tanur,hed,kuliamin,day,petrenko,vrub}@ispras.ru

Abstract. The article presents an approach to development of software standards usage infrastructure. The approach is based on formalization of standards and automated conformance test derivation from the resulting formal specifications. Strong technological support of such a process in its engineering aspects makes it applicable to software standards of real-life complexity. This is illustrated by its application to Linux Standard Base. The work stands in line with goals of international initiative Grand Challenge 6: Dependable Systems Evolution [1].

1 Introduction

The needs of economical and social development make current software systems very complex. Such a system usually consists of many components of different vendors, and a lot of individual software engineers take part in its construction. Due to this fact, interoperability of those parts and reliability of the system as a whole become problematic. The well-known way to solve these problems is enforcement of *software interface standards.*

The idea of interface standards is rather clear – we make possible for different software systems to work together through standardized interfaces without hard restrictions on the internal implementations of them. So, interoperability is ensured without damping individual creativity and corporate innovation potential. This approach works well if the standard defines the functionality of the corresponding interfaces clearly, unambiguously, and precisely.

However, looking at the current state of software standards one can see that many of them are not so clear. Historically, they were developed under the market pressure to ensure some interoperability taking into account conflicting interests of many software vendors. In such a situation, only the basic functionality can be defined unambiguously. And each group of developers usually has its own solution for peculiar and complex cases. Since such a solution is already invested, it is hard for vendors to throw it away and take the point of a competitor, which preserves its investments.

Usual compromise is to agree with some minor changes for each of competing vendors and to make the standard ambiguous in cases where serious elaboration is needed. Thus solutions in use can be declared as conforming to it.

This helps vendors to spend acceptable money to standard enforcement in their systems, but also compromise the desired interoperability of different implementations. Some of current software and telecommunication standards appeared about 20-25 years ago, and, of course, with each revision they become more strict and more consistent. Most of ambiguities of the first versions were removed, but newer additions and elaborations required by technological progress still may have unclear and equivocal statements.

The way out of this situation proposed by many software engineering researchers and practitioners is *formalization of standards.* Formal re-statement of standard's requirements discloses contradictions and ambiguities, but exacts a lot of hard work. Nobody expects that inaccuracy and inconsistency will be removed from standards at once. But by starting formalization, we at least will be aware of real problems and can make practical suggestions on their solutions.

Formalization definitely makes standards more useful, but alone it does not give instruments of their enforcement in practice. Developers of real-life systems need tools that help them to ensure conformance to standards. So, practically useful formal description of standard requirements should be supplemented with test suites checking conformance to these requirements. The approach described in this article uses one of the most elaborated frameworks for conformance testing based on formal descriptions presented in telecommunication standards ISO 9646 [2] and ITU-T Z.500 [3].

Solid formal framework makes conformance test construction more rigorous, and therefore helps to enforce quality of the systems implementing the standard. However, practical use of conformance test suites brings into account additional engineering issues.

- *Requirements traceability.* For software engineers and their managers the real source of requirements is the standard text. Formal specification is an additional document that should clearly demonstrate its correspondence to the standard. The same holds for tests derived from them – they should be traced to some requirements stated in some sections of the standard. This really helps conscientious development of standard implementations.
- *Component-wise treatment of standard.* Real-life standards are complex, as well as real-life software. Some decomposition techniques are required for adequate treatment of them. Specification formalism used should support definition of separate components of the standard, should allow their consideration in isolation, but also provide means for describing them as a whole.
- *Change management.* Standards and their implementations are not fixed entities – they are changing. These changes should have adequate support in the process of specification development, test derivation and translation. Lack of this support quickly makes the specifications and tests useless.
- *Configurations.* Real-life standards describe parameterized systems, having a lot of configuration options that significantly influence their functionality. Such options should be also supported in specifications and tests.

All those problems should have suitable solutions in the technology that aspires to provide an adequate infrastructure for standard enforcement. This article presents a candidate approach based on UniTesK technology of automated test construction developed in ISPRAS. Main ideas and techniques of the approach are considered in the next two sections. The fourth section describes preliminary results of its application to Linux Standard Base [4], the industrial standard on interfaces of Linux operating system core libraries. The last two sections of the article contain brief comparison of the approach presented here with other methods and a conclusion stating main results and directions of future development.

2 Standard Formalization

The main difficulties of standard formalization are concerned with informal and compromise nature of the industrial standards. Their main goals are to fix the consensus of main vendors on functionality of related systems and to provide reference and programming guide for developers of both the standard implementations and the systems using them. They use the language and the structure that seem to be suitable for these goals.

Interface standards often consist of two parts – rationale, presenting the main concepts and features in their integrity, and reference, describing each interface element separately. In addition to interface elements (data types, operations, constants, global data elements) reference can describe complex entities – large subsystems, header files, etc. Reference sections can refer to each other and contain parts of each other.

Standard formalization consists of several activities.

1. *Standard decomposition.* Since the number of interface elements described in the standard can be very large, on the first step they are partitioned into logically related groups. Such a group usually is closed under operation inversion and consists of operations and types concerned with one feature. For example, operations to open and close files, to create and destroy objects should be put in one group. All further work is performed mostly inside such groups.
2. *Requirements arrangement.* The next task is to extract all standard requirements to interface elements that can be interpreted in formal way and can be checked. Since one thing can be described in several places, all the corresponding sections should be read attentively, phrase by phrase, and all the constraints found should be marked. Then, these constraints are united in some consistent set.
   It is rather tedious work, but not a mechanical or trivial one. Transition from informal to formal is essentially informal itself (M. R. Shura-Bura).
   Standard statements in different parts can be inconsistent, ambiguous, represent similar, but not the same ideas. The standard itself is not enough to make consistent conclusions. One should use for that common knowledge, related books and articles, solutions in use, and communication with authors of the standard, experts in the domain or experienced developers.
   Some aspects are made intentionally unspecified to make implementations of different vendors conforming to the standard. Amusing example of such

statement is in the current version of POSIX standard [5]. Description of `fstatvfs()` and `statvfs()` functions from `sys/statvfs.h` header says: "It is unspecified whether all members of the `statvfs` structure have meaningful values on all file systems."
There are two possible ways to resolve such situations.

– Do not check anything. In this case no constraints are extracted from the corresponding text of the standard.
– If there are few possible implementations, they all can be presented as a parameterized constraint. A configuration option corresponding to the kind of implementation used is added. The constraint to check is selected depending on this option.
  Example of this case from POSIX description of `basename()` function: "If the string pointed to by path is exactly `"//"`, it is implementation-defined whether `'/'` or `"//"` is returned."

This activity is performed until all the text of the standard concerning the chosen group of interface elements is partitioned into requirements that can be checked and other phrases that do not contain testable restrictions. Main results of this work are the following.

– *Catalogue of requirements.* It lists the requirements imposed by the standard and maps each requirement to the corresponding piece of standard's text. One requirement usually corresponds to logically complete piece of text expressing one constraint on an interface element or data element. Further this catalogue helps to ensure test adequacy in terms of original text of the standard.
– *Defects of the standard and notes* presenting found ambiguities, inconsistencies, unintentionally unclear and imprecise statements, incomplete descriptions of functionality, etc.

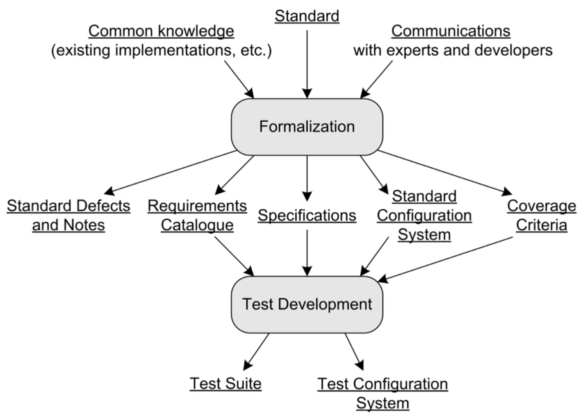

Fig. 1. Ins and outs of formalization and conformance test development process.

3. *Specification development.* This work is usually performed in some mix with the previous one. They are separated only for clarity reasons.

All the constraints found are recorded in the form of *contract specifications* of operations, data types, and interface data elements. Each operation is described with its precondition and postcondition. *Precondition* of an operation defines its domain. *Postcondition* defines the constraints on the operation results depending on values of parameters and internal system state. Data types and data elements are described with their integrity constraints called *invariants*.
This activity produces two results.
   - *Specifications* of all interface elements. Code of specifications is marked out to map the formal constraints specified to the corresponding requirements from requirements catalogue.
   - *Configuration system of the standard.* This system consists of a set of configuration parameters declared in the standard and additional ones, dependencies between them, and their links with interface elements and constraints. Some configuration options represented as values of the parameters can govern the set of constraints that should be checked. Others can say that some functionality is absent and the corresponding operations should not be called at all. Third can influence possible error codes returned by operations.
     Example of the first case is given by POSIX description of `pthread_create()` function: "If `_POSIX_THREAD_CPUTIME` is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero."
4. *Coverage criteria definition.* The last activity is to define coverage criteria that can be used to measure the adequacy of the conformance testing of standard's implementation. Basic criterion is to cover all the standard's requirements applicable to the current configuration of the system under test. More elaborate criteria can specify additional situations to be tested. All these criteria are based on the structure of specifications representing standard requirements.
   Coverage criteria can be in complex relations with configuration parameters. Possibility to cover some situation depends on values of configuration parameters and this dependency should be recorded to prevent confusion and burdensome reviews during analysis of test results.
   This work results in *a set of coverage criteria* correlated with the configuration system.

The process of standard formalization and further test construction is illustrated by Fig. 1.

3 Conformance Test Construction

The base of the conformance test construction is UniTesK technology developed in ISPRAS. It uses almost the same general formal testing framework that was developed in works of Bernot [6], Brinksma, and Tretmans [7, 8] and described in the standards ISO 9646 [2] and ITU-T Z.500 [3]. Main elements of this framework can be formulated as follows (Fig. 2 illustrates relations between them).

– The standard requirements are represented as a model in some formalism. This model is called *a specification.*
– The software system, which conformance with the standard we need to check (called *system under test*, or SUT), is supposed to be adequately modeled in the same formalism. The corresponding model is called *an implementation.* We do not know it, but assume that it exists. 'Adequate' modeling here means that we cannot observe any difference between the real behavior of the SUT and the model behavior of the implementation.

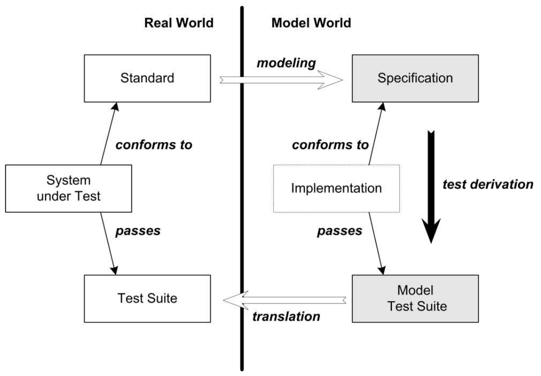

Fig. 2. Relations between real-life and model entities.

– The fact that the SUT conforms to the standard is modeled by formal *implementation relation* or *conformance relation* between the implementation and the specification.
– Model tests are derived from the specification. *A model test* is a model that provides boolean verdict as the result of interaction with other models. Implementation *passes* a test, if the latter provides the verdict **true** after this interaction. It is reasonable to derive only *sound tests,* which are passed by any implementation conforming to the specification. One wish to construct a set of model tests, or *a model test suite* that is *complete* in the sense that a model passes it if and only if this model conforms to the specification.
– Model tests are represented as test programs interacting with the SUT. Since this interaction is supposed to be modeled by the interaction between model tests and the implementation, one can conclude that the SUT passing a complete test suite conforms to the standard.

Real-life standards usually describe rather complex systems. Specification of such a standard is also complex and huge, so any complete test suite for it contains infinitely many tests. To make formal testing possible in this setting, we introduce additional element of the framework.

– *A coverage criterion* is an equivalence relation on model tests. We use coverage criterion with the corresponding hypothesis stating that an implementation either passes any two pair of equivalent tests or does not pass both of them. The criterion is chosen in such a way that it is reasonable to consider only such implementations. A test suite is called *complete according to the coverage criterion* if the union of equivalence classes of tests of this suite is a complete test suite in the traditional sense.

Note that coverage criterion can be taken from different sources. The only desired property is existence of finite test suite complete according to the criterion used. In conformance test construction we use criteria derived from the structure of specifications.

Here we present only basic ideas of UniTesK, details can be found in [9–12].
.

- Functional requirements to the SUT's behavior are stated in the form of *contract specifications.* Contract of a component consists of pre- and postconditions of all its operations and asynchronous events provided and invariants of its internal data.
- *Test coverage criteria* are defined on the base of postcondition structure. Examples of such criteria are functional branch coverage and multiple condition coverage of postconditions. One can add user-defined criteria.
- *Test scenarios* are targeted to achieve certain level of coverage of some group of operations. Such test scenario describes a finite state machine (FSM) modeling the component's behavior in such a way that its transition tour ensures 100% coverage according to the criterion. It defines state calculation procedure and a set of applicable actions for an arbitrary state. Each action corresponds to call of some operation with some arguments.
- Tests are generated during test execution by on-the-fly automatic construction of paths on the FSM described by a test scenario.

Conformance test construction for real-life software standards also requires development of *a test configuration system.* This system includes standard configuration system and additional parameters that influence test execution process. They correspond to different target coverage criteria, different sets of test scenarios, and so on. Powerful configuration system makes the resulting test suite useful in any settings where an implementation of the corresponding standard can operate in practice.

4 Applications of the Approach

The approach described in the previous sections was used for formalization and test construction for parts of IPv6 and IPMP2 protocol standards [10, 13].

More serious case study is provided by the project of Linux Verification Center on formalization of Linux Standard Base (LSB) [4]. The goal of this project is to create formal specifications of LSB requirements and corresponding conformance test suite for core libraries of Linux operating system described in the sections III and IV of LSB 3.1 and including 1532 functions. The requirements stated there in many cases (but not always!) coincide with POSIX [5] requirements to the same functions.

Now the project is in progress. Its results will be accessible at the end of 2006 as open source and will include the following.

– The set of notes to the text of LSB standard, pointing out unclear, inconsistent, or ambiguous statements.
– Parameterized conformance test suite (including formal specifications with mapping from them to the standard requirements). The parameters will control configuration of the SUT, testing quality level, test execution time, and so on.

On the first step, 1532 functions were partitioned into 147 groups, which in turn are grouped into larger subsystems according to the features concerned – threads, interprocess communication, file system, memory management, mathematical functions, etc. During first three months of the project 321 functions of 41 groups were completely specified and supplemented with base level test scenarios. About 2150 separate standard requirements were extracted for them. Some functions have just a few corresponding requirements, while others – several dozens.

The productivity achieved shows that the project will require about 15 man-years for complete formalization (with development of basic level tests) of LSB. An experienced programmer (not an average tester!) can be trained to a level needed by this project in about a month. These intermediate results give hope that the approach presented is able to cope with tasks of such a size.

The preliminary results obtained also demonstrate rather high quality of the tests developed. Tests are targeted mostly to cover all the requirements extracted during standard formalization (more presicely, those of them that are achievable on the configuration under test). At the same time they achieve high levels of source code coverage. For example, tests for 24 functions working with threads get 72% coverage of GLIBC source code lines, more than most analogous test suites (LTP test suite [14] gets 48%, LSB binary conformance test suite [15] gets 71%). Only specialized GLIBC test suite [16] gets more (78%), since it is written by the developers knowing peculiarities of the library and paying no attention to check strict conformance to the standard. For 41 functions working with strings the numbers are similar: our test suite gets 91%, LTP – 53%, LSB conformance test suite – 67%, GLIBC test suite – 84%. For 13 utility search functions (located in `search.h`) our test suite gets 65%, LTP – 28%, LSB conformance test suite – 33%, GLIBC test suite – 70%.

5 Related Work

Most advances in standard formalization and conformance test construction techniques were made in telecommunication domain. The general framework [3, 6, 8] (see also above) was developed in this area.

Our approach has a lot in common with it. The differences are related with larger size of OS interface standards such as POSIX or LSB in comparison with typical protocol standards. Scalability considerations lead to introduction of coverage criteria in the general formal testing framework and coverage-oriented test generation. Use of contract specifications makes possible more suitable decomposition then automata or labeled transition systems used by most research and industrial development in telecommunications.

Article [17] presents another attempt of standard formalization on the example of IEEE 1003.5 – POSIX Ada Language Interfaces. Standard requirements were transformed there into formally described tests directly, without intermediate specifications. This method seems to us only slightly different from manual test development.

More close to our approach is the paper [18] presenting case studies in model based testing with GOTCHA-TCBeans toolkit. One of those case studies is concerned with testing POSIX function `fcntl()`. The methodology of standard formalization used in this work is focused more on getting effective formal model for testing than on traceability to standard requirements, as in our case. Nevertheless, the approach presented in [18] uses very similar ideas with our approach, although it is based on other kind of formalism – FSMs described in Murphy language.

There are a number of similar activities of conformance test suite development for operating system interface standards. Most well-known standards in this field are IEEE Std 1003.1 (POSIX) [5] and Linux Standard Base (LSB) [4]. The main conformance test suite development projects dealing with them are the Open POSIX Test Suite [19], an open source project, and official certification test suite for LSB conformance [15] developed by Free Standards Group [20].

Both these projects use similar techniques for requirements extraction from standard text and requirements catalogue creation. Then, tests are developed manually using traditional approaches, one test per requirement. They do not perform formalization of requirements and do not use automated test construction techniques.

Note that the approach 'one test per requirement' tempts to consolidate many separate constraints into one requirement to be able to check it within one test. In Open POSIX Test Suite we found examples of requirements that correspond to dozen of the requirements extracted in our project. This ends up in situations when sub-constraints of big requirement are passed over and not tested while the whole requirement is reported as tested. So, the resulting reports can be too optimistic on the coverage of the standard requirements.

Automated test derivation from specifications used in our approach makes test suite much more manageable in case of changes in the standard. It forces to record one piece of knowledge in one place.

6 Conclusion

Enforcement of software interface standards usage is a complex task having technical, economical, and social aspects. Nevertheless, all experts agree on its necessity for stable development of software industry. Standard formalization was proposed a lot of times as a possible way to solve at least numerous technical problems related to this activity. However, formalization requires a lot of effort and many doubts were expressed on its applicability to software standards of real-life size and complexity. The project mentioned in the Case Studies section seems to be one of the first attempts to formalize a significant part of industrial software standard and to taste the fruits of this work.

We believe that the approach presented allows successful accomplishment of projects of such a scale. Strong arguments in favor of this opinion is given by the stable progress of the project, the history of its technological background (see [11, 13]), and choice of good engineering practices as a main guide [12]. Taking into account real-life engineering and organizational issues helps to avoid producing ungainly and useless results usually pertinent to pioneering use of advanced methods in practice.

We consider this work as a part of effort concerned with Dependable System Evolution Grand Challenge [1]. Tony Hoar suggested two tracks of activities dealing with this problem: development of methods and tools capable to help in resolving it and development of “challenge codes” – realistic examples of usage of those methods and tools. The last track is necessary to demonstrate ways from state of the art in software engineering to state of the practice, to find the scope of systems, for which advanced the methods developed in research community are applicable.

To stimulate activities of the second kind, ISPRAS donates the results of the project and all the tools needed to deal with them to open source community. We hope that members of this community can be attracted to more challenging projects such as formalization of the huge set of Carrier Grade Linux standards [21], embedded and real-time versions of Linux, and some widely used programming languages.

Acknowledgements. We thank Federal Agency for Science and Innovations of Russia for its financial support of the Linux Verification Center creation and LSB formalization project.

References

1. http://www.fmnet.info/gc6/
2. *ISO 9646. Information Theory – Open System Interconnection – Conformance Testing Methodology and Framework.* ISO, Geneve, 1991.
3. *ITU-T. Recommendation Z.500. Framework on formal methods in conformance testing.* International Telecommunications Union, Geneve, Switzerland, 1997.
4. http://www.linuxbase.org/spec
5. http://www.unix.org/version3/ieee_std.html

6. G. Bernot. *Testing against Formal Specifications: A Theoretical View.* In Proc. of TAPSOFT'91, Vol. 2. S. Abramsky and T. S. E. Maibaum, eds. LNCS 494, pp. 99–119, Springer-Verlag, 1991.
7. E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. *A formal approach to conformance testing.* In J. de Meer, L. Mackert, and W. Effelsberg, eds. 2-nd Int. Workshop on Protocol Test Systems, pp. 349–363. North-Holland, 1990.
8. J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.
9. I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. *UniTesK Test Suite Architecture.* In Proc. of FME 2002. LNCS 2391, pp. 77–88, Springer-Verlag, 2002.
10. V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev, I. Bourdonov. *Integration of Functional and Timed Testing of Real-time and Concurrent Systems.* In Proc. of PSI 2003, LNCS 2890, pp. 450–461, Springer-Verlag, 2003.
11. V. Kuliamin, A. Petrenko, A. Kossatchev, I. Bourdonov. *UniTesK: Model Based Testing in Industrial Practice.* In Proc. of 1-st European Conference on Model-Driven Software Engineering, Nurnberg, December 2003, pp. 55–63.
12. V. Kuliamin. *Model Based Testing of Large-scale Software: How Can Simple Models Help to Test Complex System.* In Proc. of 1-st International Symposium on Leveraging Applications of Formal Methods, Cyprus, October 2004, pp. 311–316.
13. V. Kuliamin, A. Petrenko, N. Pakoulin. *Practical Approach to Specification and Conformance Testing of Distributed Network Applications.* In M. Malek, E. Nett, N. Suri, eds. Service Availability. LNCS 3694, pp. 68–83, Springer-Verlag, 2005.
14. http://ltp.sourceforge.net/
15. http://www.linuxbase.org/download/#test_suites
16. ftp://ftp.gnu.org/gnu/glibc/
17. J. F. Leathrum and K. A. Liburdy. *A Formal Approach to Requirements Based Testing in Open Systems Standards.* In Proc. of 2-d International Conference on Requirements Engineering, 1996, pp. 94–100.
18. E. Farchi, A. Hartman, and S. S. Pinter. *Using a model-based test generator to test for standard conformance.* IBM Systems Journal, 41:89–110, 2002.
19. http://posixtest.sourceforge.net/
20. http://freestandards.org/
21. http://www.osdl.org/lab_activities/carrier_grade_linux