Generating Reliable Conformance Test Suites for Parallel and Distributed Languages, Libraries, and APIs

Łukasz Garstecki[1,2]

[1] Faculty of Electronics, Telecommunications and Informatics,*
Gdańsk University of Technology, ul. Narutowicza 11/12, 80-952 Gdańsk, Poland
galu@eti.pg.gda.pl
[2] Laboratoire Informatique et Distribution**, ENSIMAG - antenne de Montbonnot,
ZIRST, 51, avenue Jean Kuntzmann, 38-330 Montbonnot Saint Martin, France
Lukasz.Garstecki@imag.fr

Abstract. This paper outlines a new methodology for generating Conformance Test Suites (CTS) for parallel and distributed programming languages, libraries and APIs. The author has started his research in the field of conformance testing for parallel data-driven language Athapascan, invented a methodology for designing and analyzing CTSs called Consecutive Confinements Method (CoCoM), developed a tool called CTS Designer, which implements standard ISO/IEC 13210 and the CoCoM methodology, and finally created a CTS for the crucial part of the Athapascan language. Although CoCoM was originally meant for parallel and distributed software, it is not limited only to it and can be also used for a variety of other languages, libraries and APIs.

Keywords: Software testing, parallel data-driven languages.

1 Introduction

Conformance testing is a black-box testing and is aimed at demonstrating that an implementation, called Implementation Under Test (IUT), fulfills its specification. Conformance testing is very well known for example in the domain of protocols and communication systems, where IUT can be relatively easy represented by a Finite State Machine. Conformance testing of programming languages, libraries and APIs requires however different solutions and the only one standard directly connected to this issue found by the author is ISO/IEC 13210 [1]. Although dedicated to POSIX, this standard is quite general and can be applied to different programming languages, libraries and APIs. In the paper we utilize the notions provided by it. The specification of an IUT is called a *base standard* and basic entities in the base standard, such as functions, constants, header files, etc.

* Funded in part by the State Committee for Scientific Research (KBN) under Grant 4-T11C-004-22.
** Laboratory is funded by CNRS, INRIA, INPG and UJF.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 74–81, 2004.

are called *elements*. Each element has a number of *conformance requirements* it must satisfy. The specification for testing a requirement is called an *assertion*, and is required to be true in a *conforming implementation*. Additionally, we will provide a notion of a *potential error* of an assertion, which defines potential failures in this assertion. Standard ISO/IEC 13210 specifies how to identify elements in a base standard, and how to retrieve requirements and assertions for them, but unfortunately does not specify how to design and implement conformance tests. According to the knowledge of the author, there exist only two tools that are based on this standard[1] (apart from CTS Designer developed by the author): TET [2] developed by The Open Group and DejaGnu [3] developed by Free Software Foundation. Although these tools are well established and very useful, and moreover there exists even a special version of TET meant for distributed software, the author revealed a serious problem that can appear in Conformance Test Suites (CTSs), and which is not considered by these tools and by any solution known to the author. The problem lies in relations between tests in a CTS, and can result in an unreliable CTS, i.e. a CTS that can pass for faulty IUTs. The problem presents in Sect. 2, while Sect. 3 specifies precisely conditions under which the problem appears. Next, Sect. 4 is presenting a methodology called *Consecutive Confinements Method* (CoCoM), which solves the problem. Finally, Sect. 5 contains conclusions and plans for the future work.

The CoCoM methodology, presented in the paper, can be seen as a complement of existing solutions, or in other words as a framework, where different methodologies and tools could be used. However, the author developed its own prototype tool called *CTS Designer*, based on standard ISO/IEC 13210 and methodology CoCoM, and finally created a CTS for the crucial part of a parallel data-driven language called Athapascan [4] using his tool.

For the sake of simplicity and clarity we will refer throughout the paper to programming languages, libraries and APIs shortly as "programming languages", but the presented methodology applies to any of them equally. Also wherever we use the term "conformance testing", we mean conformance testing for programming languages, libraries and APIs.

2 Unreliable Conformance Test Suites

As we have written at the beginning, conformance testing is black box testing, so very often the only one way to test an element is to use another elements under tests. This takes place especially in a case of parallel and distributed programming languages, where usually the only one way to test a function sending a message is to use another function that is receiving a message, since in conformance testing we cannot access a source code of an IUT. The problem is, that afterward very often to test the receiving function, we will use the sending function, that we have tested just before. And when we use a function in a test, we

[1] Actually there are based on the former version of this standard, namely ISO/IEC 13210-1994, IEEE 1003.3-1991.

implicitly assume that its implementation is valid. So using words of mathematics, first we prove A, assuming that B is correct, and then we prove B assuming that A is correct. Below, we present a small example showing what can happen, if we do not care about this problem. Elements used in this example are: *write* (denoted as `W`), *read* (`R`) and *access* (`A`). These are three methods taken from Athapascan [4] that relatively write, read and access[2] a shared variable. For the sake of clarity let us assume that the shared variable is an integer, denoted as `I`. Now, let us assume that our system consists only of two elements: `W` and `R`, and let us assume that `W` always writes one more (+1), and `R` always reads one less (-1). The situation is presented in System 1 in Fig. 1.

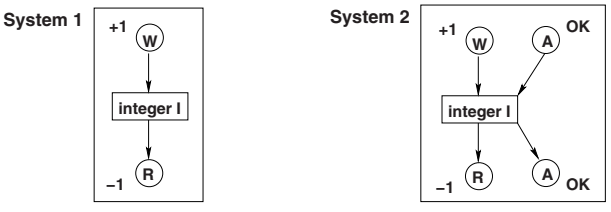

Fig. 1. Simple systems with `read`, `write` and `access`

Now, we can pose a question: *Does System 1 contain a functional error?* It can be sure that there is no failure, because failures occur when provided services are not conforming to their specifications [5], which is not the case here. May be we can say that there is a defect, however it is also not so sure because it could be a conscious decision of system designers, because for example storing zero value could be very costly or even impossible (especially in mechanic, not computer systems). However, regardless of whether we consider that this is a defect or not, we do not have any observable error here, because provided services will be always conforming to their specifications. Now, let us slightly change our system and let us add element `A` and let us assume that it works correctly. The situation is presented in System 2 in Fig. 1.

Now, our system will obviously contain failures, because for example if we accessed integer `I` previously written by `W`, we would obtain a wrong value of `I`. That means that the same implementations of elements `W` and `R` can be now evaluated as non-conforming to their specifications. "Can be" because by observing only responses of this system, we are not able to say which elements are faulty, because the same symptoms would appear, if for example element `A` always subtracted 1 when writing, and added 1 when reading. Luckily, in conformance testing, the principal question is whether **all** elements conform to their specification or not. In other words, we don't have to find which elements are faulty, but whether at least one element is faulty and no matter which.

[2] That means they can read it and write it.

So, in these two simple examples we could see that the same implementations of elements `W` and `R` can be once evaluated as conforming to their specification and the other time as non-conforming. This pushes us toward a very important conclusion, namely that conformance is relative, i.e. conformance of an element to its specification, depends not only on the element itself, but also on the *context of this element*, i.e. other elements either from IUT or out of IUT. If we do not care about the context of an element, we can obtain for example a CTS, presented in Fig. 2, which would pass for the faulty implementation of System 2 presented in Fig. 1.

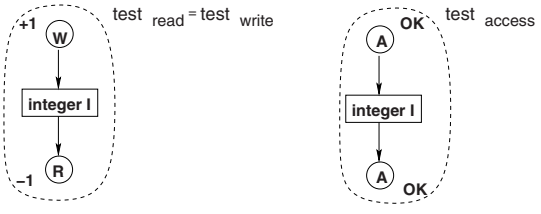

Fig. 2. A simple CTS for System 2 in Fig. 1, passing for an invalid IUT

In the next section we will say when the context of an element has to be considered and which elements from this context should be taken into account.

3 Dynamic Correctness of a Test

As we have written before, very often we can test a given element from IUT only by means of other elements, which are under test too. But, if an element is under test, we cannot be sure whether this element is valid or not (i.e. whether its implementation is valid or not) unless we do appropriate tests. Meanwhile, traditionally we implicitly assume that elements used in a test are valid. This pushes us to introducing two new terms: *static* and *dynamic correctness* of a test. Test is *statically correct*, if that test (namely its source code) is correct (i.e. it passes for a good implementation of an element under test and fails for an implementation that contains a potential error, for which the test was designed) under assumptions that all elements used in the test are valid. Test is *dynamically correct*, if the running test (namely its executable) is valid. In other words, by static correctness of a test we mean traditional correctness of a test. In turn, to prove that a test is dynamically correct, we should prove that it is *statically correct* and all elements used in this test are valid[3]. But to

[3] It is also worth mentioning here, that elements could be hardware too. For example, network with a very low performance could give untrue results, so it is a good idea to do performance tests of the network before we start conformance testing of parallel and distributed software [6].

prove that elements used in a given test are valid, we need dynamically correct tests that will test them. In that way, we can build a directed graph $G(V, E)$ of dependencies between tests, where tests are represented by nodes and directed edge E_{ij} indicates that test $\mathtt{T}_i$ needs test $\mathtt{T}_j$ to prove its dynamic correctness. We will say that test t_i *directly depends on* test t_j. Moreover, if there exist tests $t_k, t_{k+1}, \ldots, t_{k+n}$, where $n > 0$, such that $\forall_{l=\{k,\ldots,k+n-1\}} t_l$ *depends on* t_{l+1}, we will say that test t_k *indirectly depends on* test t_{k+n}. Finally, if t_i depends on t_j and t_j depends on t_i, we will say that test t_i *has direct cyclic dependence with* test t_j. We can also say that if t_i indirectly depends on t_j and t_j indirectly depends on t_i, then test t_i *has indirect cyclic dependence with* test t_j. In the farther part of this paper we will shortly say: *depends on* instead of *indirectly depends on* and *cyclic dependence* instead of *indirect cyclic dependence*. In the directed graph $G(V, E)$ cyclic dependence manifests as a cycle.

However, errors of elements used in a test do not affect the dynamic correctness of the test always in the same way. We should distinguish three different relations between error b_i of element e_i, for which test t_i is designed, and error b_j of an element e_j used in test t_i.

1. *masking error* – error b_j can *mask* b_i, i.e. test t_i can pass for an invalid implementation of element e_i.
2. *triggering error* – no masking error, but error b_j can *trigger* b_i, i.e. test t_i can fail for a valid implementation of element e_i.
3. *no relation* – error b_j has no influence on the dynamic correctness of test t_i.

By analyzing different cases one by one, we could find that the overall test result of a CTS can be wrong only if it contains at least two tests t_i and t_j which have cyclic dependence, and errors that are to be detected by these two tests can mask each other. We will say then that tests t_i and t_j have *strong cyclic dependence*. Notice for example, that in Fig. 2 non-conforming IUT would be evaluated as "conforming", because there is strong cyclic dependence between tests $test_{write}$ and $test_{read}$. In the rest of situations, even if a CTS contains dynamically incorrect tests (what means that at least one of elements is not valid), then single tests can return incorrect test results, but we could show that at least one test from the CTS will fail, and thus the whole CTS will fail, because in conformance testing IUT conforms to its base standard only if all tests pass.

The obvious conclusion from our considerations is that we should get rid of strong cyclic dependencies if we want a reliable CTS. The easiest solution is to find another test scenarios using another elements. However it can happen that we have already used all possible elements, that are able to test a given element e_i, and in each of those tests we will have strong cyclic dependencies. In such a situation, we should run all these tests. If all these tests pass, that means that either there is no error in element e_i or that there are errors, but these errors are not observable. In such a situation we will say that the tests for element e_i are *dynamically p-correct* (p comes here from *partially*). Naturally, if some other tests depend on dynamically p-correct tests, they can not be supposed to be dynamically correct. Examples of dynamically p-correct tests can be tests *write-read* for `W` and `R` in System 1 presented in Fig. 1. P-correct tests can frequently

take place especially in black box testing of parallel and distributed software, where very often the only one way to check sent or received messages, or to examine the state of shared variables is to use another elements from the IUT.

4 Consecutive Confinements Method

Now, we are ready to give the algorithm for constructing a reliable CTS, i.e. a CTS which will pass for valid IUTs, i.e. implementations that fulfill all requirements from their specification, and will fail for invalid implementations, i.e implementations that contain at least one potential error defined for at least one assertion of at least one element, assuming that this error is observable.

Algorithm 1 *Design a conformance test suite for a base standard*

1. *Find all elements in the base standard and retrieve assertions (from a base standard) for them, using standard ISO/IEC 13210 [1].*
2. *Identify implicit preconditions in assertions.*
3. *For each assertion, enumerate potential errors.*
4. *For each potential error find at least one test scenario.*
5. *For each test scenario in the CTS make analysis using Algorithm 2.*
6. *Take any test t (for potential error b of assertion a of element e) from the CTS such that t has strong cyclic dependence with at least one another test in the CTS. If such a test does not exist, then finish.*
7. *If possible, write a new version of test t for potential error b of element e, using different elements (than in the previous versions of the test) that are able to test element e. Otherwise, i.e. if no more tests can be found, include all tests found for error b into the CTS, and mark these tests as dynamically p-correct.*
8. *Go to Step 5.*

For the sake of clarity, we have skipped some implementation, but important details. For example, each created test t for a given potential error shall be kept in a memory, even if t is dynamically incorrect and was replaced by another dynamically correct test t', because dynamic correctness is not a feature, which is constant (in contrary to static correctness). For example, it can happen later, that another test t_z will be added to CTS, such that t' will have a strong cyclic dependence with t_z, and thus t' will become dynamically incorrect. In such a situation, we will have to look for another test, alternative to t'. Then, first of all we have to know that test t was already designed, and moreover if it happens that no other alternative tests can be found, both tests t and t' will be necessary.

In Step 4 and 7 many different methodologies and tools for designing test scenarios can be used. This is why we wrote that CoCoM can be seen as a framework. Algorithm 2 for making analysis of relations between a given test t and all other tests in a CTS, which is used in Step 5, is presented below. In Step 8 we come back to Step 5, where we will remake the analysis for all test scenarios. We have to do it, because after adding a new test to a CTS, theoretically each

test in a CTS can begin to have strong cyclic dependence with the newly added test (directly or indirectly). However, if we store the state of the whole analysis in memory, then only analysis for the newly added test must be done manually, while the analysis for all other tests can be easily done automatically (using the information provided earlier in Step 1 of Algorithm 2), like it is for example in the case of the prototype tool CTS Designer, developed by the author.

Algorithm 1 gives the answer for the question how to combine different elements into tests to obtain a reliable CTS, i.e. when it is enough to create only one test scenario to detect a given potential error of an element, and when we have to create more test scenarios using different elements from IUT. Below we present Algorithm 2 for the analysis of relations between tests.

Algorithm 2 *Make analysis of relations between test t and all tests in CTS*

1. *For each element e_x used in test t do*
   a) *Describe briefly a function of element e_x in test t.*
   b) *For each assertion a_x of element e_x do*
      i. *If assertion a_x is not used in test t, then go to Step 1b.*
      ii. *For each potential error b_x of assertion a_x do*
         A. *If error b_x cannot mask error b, then go to Step 1(b)ii.*
         B. *For each test t_x for potential error b_x, mark that test t depends on test t_x.*
2. *Find cycles (direct and indirect) in graph of dependencies between tests.*

So, as we can see, by the *consecutive confinements* we can narrow the group of tests, with which a given test t can have strong cyclic dependence. That is why the methodology is called Consecutive Confinements Method – shortly CoCoM.

Step 1a was provided to facilitate further decisions in Step 1b, whether a given assertion a_x is used in test t or not. Steps 1(b)iiB and 2 can be naturally done automatically. The rest of points must be done manually if no supporting formalisms are used[4]. However, it is worth mentioning that we can also reduce the amount of manual analysis by excluding from analysis cases, where cycles can not happen. For example, if element e_x (used in test t for element e) does not have any test, which uses element e (directly of indirectly), there is no point in analyzing relations between test t and assertions of element e_x, because cycles will not appear. In the similar way, we can also reduce analysis at the level of potential errors. It is also worth mentioning that Step 2 can be done "in fly", i.e. new cycles can be searched after any changes in the state of analysis made in Step 1. This takes place for example in CTS Designer.

5 Conclusions and Future Work

The paper presented a problem because of which a CTS can be unreliable, i.e. it can pass for invalid IUTs. The problem can appear if elements use themselves

[4] Notice, that CoCoM does not put any constraints on notations used for expressing assertions and errors, so they can vary from the natural language to formalisms that allow for automatic generation of tests.

to test each other. The paper precisely specified situations where CTSs can be unreliable and proposed a methodology for generation reliable CTSs, called CoCoM. The methodology also gives clear criteria when a given element can be tested by only one another element (if it is necessary), and when it requires to be tested by a group of different elements, what allows for optimizations of a CTS. The methodology was proved to be useful for parallel and distributed languages by designing a CTS for the crucial part of parallel data-driven language Athapascan by means of the prototype tool CTS Designer, developed by the author on the basis of standard ISO/IEC 13210 and the CoCoM methodology. CTS Designer works currently under MS Windows, and generates CTSs ready to execute on Linux. The results of the executed CTS can be viewed as XHTML pages. In the nearest future, the author plans to make experiments with other parallel and distributed languages, libraries and APIs, like PVM and MPI.

Acknowledgments. I wish to thank Bogdan Wiszniewski from Gdańsk University of Technology and Jean Louis Roch and Jacques Chassin de Kargommeaux from L'Institut National Polytechnique de Grenoble for their helpful suggestions and comments on my work, and their cooperation in conformance testing of Athapascan language.

References

1. ISO/IEC: ISO/IEC 13210:1999(E), IEEE Std 2003, Information Technology — Requirements and Guidelines for Test Methods Specifications and Test Method Implementations for Measuring Conformance to POSIX Standards. (1999)
2. The Open Group `http://tetworks.opengroup.org/documents/docs33.html`: Test Environment Tool – TETware User Guide. Revision 1.4 edn. (2003)
3. Savoye, R.: Deja Gnu – The GNU Testing Framework. Free Software Foundation, `http://www.gnu.org/software/dejagnu/`. 1.4.3 edn. (2002)
4. Roch, J.L., Galilée, F., Doreille, M., Cavalheiro, G., Maillard, N., Revire, R., Defrenne, A.: Athapascan : API for Asynchronous Parallel Programming, `http://www-id.imag.fr/ Logiciels/ath1/manual`. (2002)
5. Laprie, J.C.: Dependability: Basic Concepts and Terminology. Springer-Verlag (1991)
6. Garstecki, Ł., Kaczmarek, P., Krawczyk, H., Wiszniewski, B.: Conformace testing of parallel languages. In Kacsuk, P., Kranzlmüller, D., Németh, Z., Volkert, J., eds.: Distributed and Parallel Systems Cluster and Grid Computing. Volume 706 of The Kluwer International Series in Engineering and Computer Science., Kluwer Academic Publishers (2002) 173–181
7. Garstecki, Ł., Kaczmarek, P., de Kergommeaux, J.C., Krawczyk, H., Wiszniewski, B.: Testing for conformance of parallel programming pattern languages. In: Lecture Notes in Computer Science, Springer Verlag (2001) 323–330
8. Garstecki, Ł.: Generation of conformance test suites for parallel and distributed languages and apis. In: Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE (2003) 308–315