Structural Analyzis of Large TTCN-3 Projects

Kristóf Szabados

Ericsson Hungary

Abstract. Experience has shown that the Testing and Test Control Notation version 3 (TTCN-3) language provides very good concepts for adequate test specification, but it was not presented yet how large test systems written in TTCN-3 are structured. This paper presents my efforts to analyze the structure of TTCN-3 projects, to see if such problems manifest, and if possible to provide methods that can detect these problems.

1 Introduction

The Testing and Test Control Notation version 3 (TTCN-3) is a high level programming language standardized by the European Telecommunications Standards Institute (ETSI) for automated black box and reactive tests. It was designed mainly for testing of telecommunication protocols and Internet protocols, but nowadays it is also used for testing aerospace, automotive or service oriented architectures. Although it was not able to get a wide community yet, several companies interested in network protocols are already using it for testing their systems.

Inside Ericsson there are projects whose amount of source code has almost reached 1 million lines. As we realized the amount of code in existence, it became a major area of interest to study its structure. When the code structure of a project becomes too complex, the maintenance costs are expected to grow as several problems start to surface.

At this point it is important to note, that in TTCN-3 the highest code structuring construct is the module of which every project is built up. TTCN-3 modules are like classes in C++ or Java (however TTCN-3 is actually closer in its philosophy to C). But unlike C++, which has namespaces and Java, which has packages to structure their classes, TTCN-3 does not provide any higher level primitives that the user could use to group their modules.

2 Motivation

The main interest is to analyze the structure of the large source code bases, to find a way to provide guidelines for projects to follow, or to compare against their measured data. Up to now such huge projects were evolving on their own without any tool to monitor them. So as a first step what was need most was some kind of visualization of their structure.

3 Experimental setup

The module structure of 9 projects was analyzed by measuring the incoming and outgoing connections of each module, and creating graphs on the collaborations between them.

A tool was written to process the semantic graph of the projects and collect statistics regarding the collaboration graphs of modules. The base for all the measurements and charts produced was the number of graph nodes and for each node the number of its incoming and outgoing connections.

Out of these 9 projects 6 are completely different, developed by standalone organizations inside Ericsson. IMSM_SIP and Simple hello are built on the load test framework of Ericsson Hungary. ETSI IPv6 is a standardized testsuite.

4 Results

I measured for each module how many others it imports (I(project)), and how many times it is imported (O(project)) by other modules. Table 1 shows for all projects the $I_{max}(project)$ (the biggest number of modules imported by the same module) and $O_{max}(project)$ (the biggest number of modules importing the same module). It is easy to see that projects having more modules are more likely to have a higher $I_{max}(project)$ and $I_{max}(project)$ values.

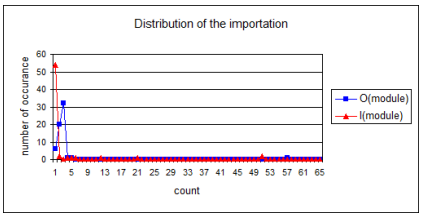
In almost all of the projects we can see that $O_{max}(project)$ is roughly between 40%-50% of the number of modules (with only CAI3G standing out considerably).

Table 1. importation data

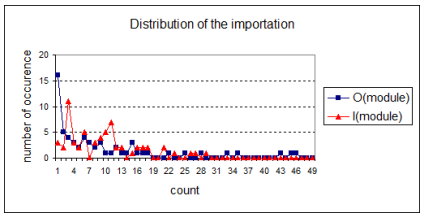
Project vs test	Number of modules	$I_{max}(project)$	$O_{max}(project)$	Linecount
TGC_traffic	20	10	6	127.470
ADC_OMA	42	23	8	21.174
CAI3G	65	51	57	53.583
ETSI IPv6	68	29	46	67.505
Test automation Wireline	71	15	34	97.672
Simple Hello (TitanSim)	124	38	49	71.751
IMSM_SIP (TitanSim)	171	49	71	79.613
W_MOCN	205	36	85	442.784
MTAS	331	78	181	794.829

4.1 Importation data analyzed by projects

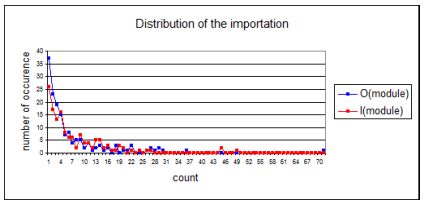
Figure 1(a) shows the distribution of $I(module)$ and $O(module)$ values for all of the modules in CAI3G. There are only a few modules that import many others,



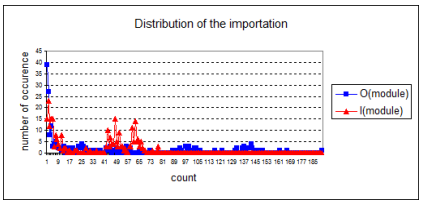
(a) CAI3G importation distribution



(b) ETSI_IPv6 importation distribution



(c) IMSM_SIP importation distribution



(d) MTAS importation distribution

Fig. 1. Distributions of importation

or are imported many times, most of the modules import only a few others, often < 5 others.

Figures 1(b) and 1(c) show that the distributions of $O(module)$ and $I(module)$ become smoother as the number of modules increases. In fact the distribution charts of both $O(module)$ and $I(module)$ values visibly converges to the shape of an exponential function with a long tail.

Figure 1(d) does not fit this schema as for both distributions, there are values which appear far more often than expected. Possibly indicating serious code organizational issues, parts of the library section of this project seems to be highly coupled.

Figure 2, that where $I(module)$ is high, $O(module)$ is usually very low, and vice versa. This implicitly indicates, that the load of the functionality seems to be distributed better among the modules, more than in any other project we have seen so far.

While checking MTAS deeply I have found that in fact the testcases (top level entry points in TTCN-3) are not concentrated in a few files, but are distributed among many. About 1/4 or 1/5 of the modules did mostly only contain testcases.

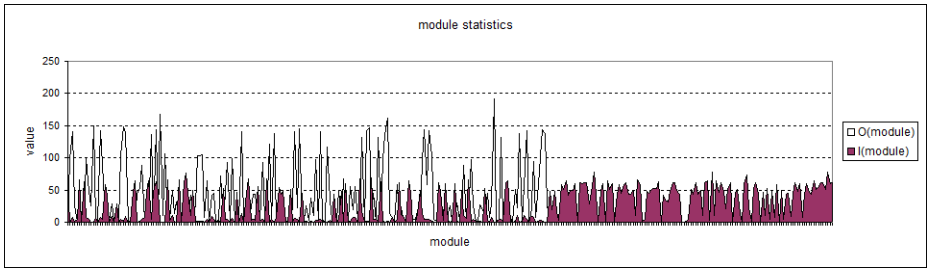


Fig. 2. $|I(module) - O(module)|$ values on MTAS, colored by the larger one

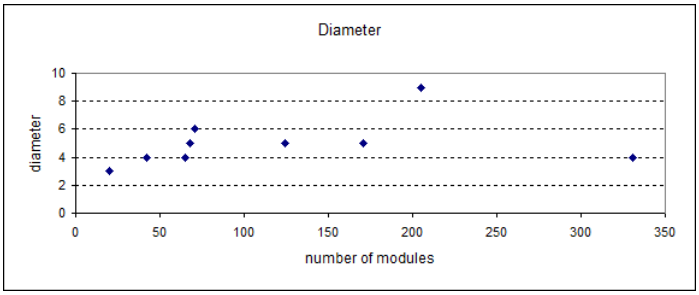


Fig. 3. Diameter of the graphs

4.2 Project diameter

Figure 3 shows one of the most surprising findings of my research. In case of TTCN-3 the diameter of the module importation graph (the longest path from the set of the shortest paths between any two nodes in the graph) does not seem to depend on the number of modules present in the project.

In TTCN-3 this means that the testers working on MTAS does not actually have to know how every little part works. They only need to understand and check the modules they are working with, and the ones depending on these. In this structure the amount of such modules is very limited.

5 Is it scale-free?

As the diameter of projects was very small, we checked whether the projects were scale-free ([2] showed, that scale-free graphs have a very small diameter).

Scale-free graphs include the physical connection forming the Internet, networks of personal contacts [6], and even the connectivity graph of neurons in the human brain [4] [5]. It was also shown, that the class, method and package collaboration graphs of the Java language [1], and the object graph (the ob-

jects instances created at runtime) of most of the Object Oriented programming languages in general [7], [3] also show scale-free properties.

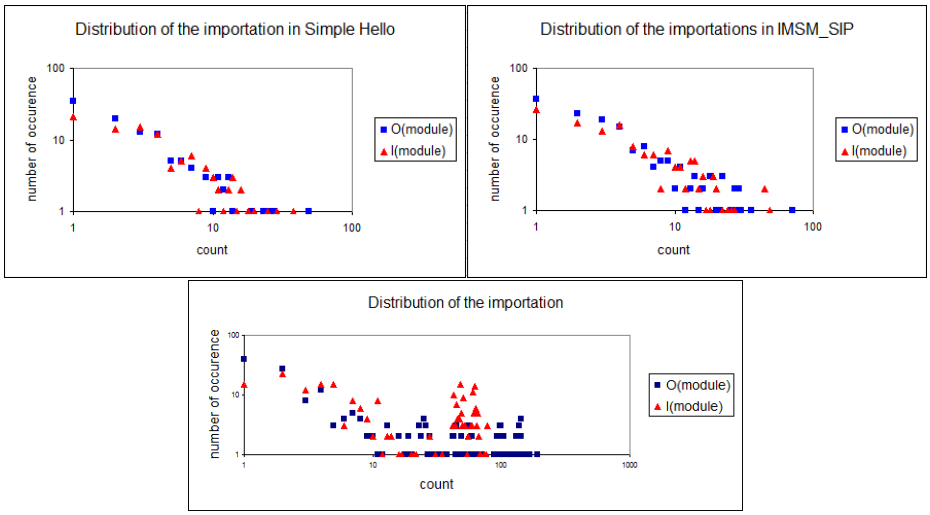


Fig. 4. Simple Hello, IMSM_SIP and MTAS on log-log scale

Figure 4 shows that both Simple Hello and IMSM_SIP displays the expected power law distribution (as a straight line with a slope), but MTAS deviates far from what we expected to see. This has to be checked with the developers of MTAS as if this is not natural in the language it might show serious structural problems.

From these data the most we can say is that TTCN-3 is, or seemingly converges to being scale-free as the language has not yet gained wide audience, leaving the number of lines of code written so far is rather small.

Being scale-free is very important as it was proven [8] that such networks have good resistance against random failures, but at the same time have an Achilles' Heel against direct attacks. The more vulnerable nodes can be detected with $I_{max}(project)$ and $O_{max}(project)$.

6 Conclusions

The measured data revealed some interesting information that had to be checked in more detail to validate that the measured data is in fact showing something of importance.

TTCN-3 projects tend to have a few modules that are imported much more often than others. These were identified to be the modules containing the type definitions, for the protocols being tested. As the language does not support

transitive importation, developers either have to import the types they wish to work with directly, or have to create wrapper functions for almost all possible values that the given types can have.

In one of the projects I have found that the $O_{max}(project)$ was way below the expected numbers. Looking into the code I have found that the developers were trying to create a layer between the data types and the actual testcases. As the messages being sent on the network were rather large, and variant, some of the functions at the time of my investigation had over 50 formal parameters.

These metrics, both the highest values and their distribution for all modules turned out to be useful:

1. A low diameter indicates very low complexity for the testers to work with
2. Values and distribution of $I(module)$ being more than expected, indicates too many direct interactions with the data, which could be improved by following better code reuse principles(like extracting very often used common parts into functions).
3. Values and distribution of $I(module)$ being much lower than expected, indicates to many layers in the software, leading to huge hierarchies of functions.
4. For modules not imported ($O(module) = 0$), their whole subgraph might not be needed in the given project, or to a given task. Tools can be provided to allow for more people to work in parallel (as random side effects will not corrupt the whole graph), and with lower build times (by filtering out un-needed code parts).
5. Having only a few modules in a project where $I(module)$ is high indicates that the functionality was not distributed evenly in the code.

References

1. Danny Hyland-Wood, David Carrington, Simon Kaplan, Scale-Free Nature of Java Software Package, Class and Method Collaboration Graphs, submitted to The 5th International Symposium on Empirical Software Engineering, September 21-22, 2005, Rio de Janeiro, Brazil.
2. Cohen R. and Hevlin, D., Scale-Free Networks are Ultrasmall, Physical Review Letters, Vol. 90, 058701, 2003.
3. de Muora A.P., Lai Y.C., Motter A.E., Signatures of small-world and scale-free properties in large computer programs, Physical review E 68, 017102 July 2003.
4. Jeong, H., Tombor, B. Albert, R., Oltvai, Z.N. & Barabási, A.-L., The large-scale organization of metabolic networks. Nature Vol. 407, pg. 651-654, 2000.
5. Barabási, A.-L. Linked: The New Science of Networks. Perseus Press, New York, 2002
6. Zipf, G. Psycho-Biology of Languages. Houghtton-Mifflin, Boston, 1935.
7. Potanin A., Noble J., Frean M., Biddle R. Scale-free geometry in OO programs. Communications of the ACM Vol. 48, issue 5, pg. 99-103, 2005
8. R.Albert, H. Jeong, A.-L. Barabasi: Error and attack tolerance of complex networks. Nature, Vo. 406, Bo. 6794. (27 Jul 2000), pp. 378-382