Search-based Testing for Embedded Telecom Software with Complex Input Structures

Kivanc Doganay[1,2], Sigrid Eldh[3,4], Wasif Afzal[2], and Markus Bohlin[1,2]

[1] SICS Swedish ICT AB, Kista, Sweden
[2] Mälardalen University, Västerås, Sweden
[3] Ericsson AB, Kista, Sweden
[4] Karlstad University, Karlstad, Sweden

Abstract. In this paper, we discuss the application of search-based software testing techniques for unit level testing of a real-world telecommunication middleware at Ericsson. Our current implementation analyzes the existing test cases to handle non-trivial variables such as uninitialized pointers, and to discover any setup code that needs to run before the actual test case, such as setting global system parameters. Hill climbing (HC) and (1+1) evolutionary algorithm (EA) metaheuristic search algorithms are used to generate input data for branch coverage. We compare HC, (1+1)EA, and random search with respect to effectiveness, measured as branch coverage, and efficiency, measured as number of executions needed. Difficulties arising from the specialized execution environment and the adaptations for handling these problems are also discussed.

1 Introduction

Embedded systems are prevalent in many industries such as avionics, railways and telecommunication systems. The use of specialized microprocessors such as digital signal processors (DSPs) and electronic control units have enabled more complex software in the embedded domain. As a consequence, quality control has become more time consuming and costly, similar to the non-embedded software domains.

It is well known that software testing is an expensive activity [6]. Thus considerable research has focused on automating different test activities, notably software test data generation. In recent years, the use of metaheuristic search algorithms have shown promise in automating parts of software testing efforts [3], including test data generation, which is commonly referred to as search-based software testing (SBST). SBST has received increasing attention in the academia. While random testing and fuzzing have gained reasonable visibility and acceptance, search-based approaches are not yet adopted in the industry. To gain wider acceptance, we believe that experiments of search-based techniques on real-world industrial software should be performed. With a family of such experiments, we would be in a better position to argue for the industrial uptake of SBST.

In this paper, we present a case study of applying search-based testing on a real-world telecommunication middleware at Ericsson. We use hill climbing

(HC) and (1+1) evolutionary algorithm (EA) metaheuristic search algorithms to generate unit level input data that exercise different branches in the control flow. The algorithms, along with random search as a baseline, are compared for effectiveness, measured in branch coverage, and efficiency, measured in number of executions. Existing test cases are automatically analyzed in order to handle complex data structures, pointers, and global system parameters. We also discuss difficulties arising from the specialized execution environment, and the adaptations for handling these problems. A further detailed account of the case study can be found in the corresponding technical report [1].

2 System Under Test

The system under test is written in the DSP-C programming language. DSP-C is an extension of the C language with additional types and memory quantifiers designed to allow programs to utilize hardware architectural features of DSPs [2]. Ericsson uses a proprietary compiler to compile DSP-C code for various DSP architectures, and a proprietary simulator to execute the resulting binaries on a normal Linux environment. The whole tool chain is connected via a test execution framework. Currently we do not integrate to the test framework, but we extract certain information from it.

2.1 Analysis and Instrumentation

We adapted `pycparser`[5], which produces the abstract syntax tree (AST) for the C language, to the DSP-C language including Ericsson specific memory identifiers. Our tool analyzes the resulting AST to produce the control flow graph (CFG) of the function under test (FUT).

The `Assimilator` module (Fig. 1) instruments the FUT by inserting observation code at the branching points in the CFG. The code is instrumented without changing its functional behavior. For example, the statement `if(a>b && c)` is instrumented as `if(observe_gt(12,0,a,b) && observe(12,1,c))`, where 12 is the branch identifier in the CFG, while 0 and 1 are the clause identifiers in the given expression. Note that the variable `c` will not be read if the first condition `(a>b)` is false. This ensures that instrumentation do not change the functional behavior of the FUT.

Existing test cases. We analyze the existing test cases (developed by Ericsson engineers) for a given FUT to discover the input interface, and any *setup code* that needs to be executed before calling the FUT. Test cases may include some setup code, such as allocating memory, or setting a global variable. We replicate these setup code sections in the generated template test case. Then all assignments in the test code are parsed, and the discovered assignments are used to define the input space. Only the variables and fields of data structures which are set in at

[5] Available at https://github.com/eliben/pycparser

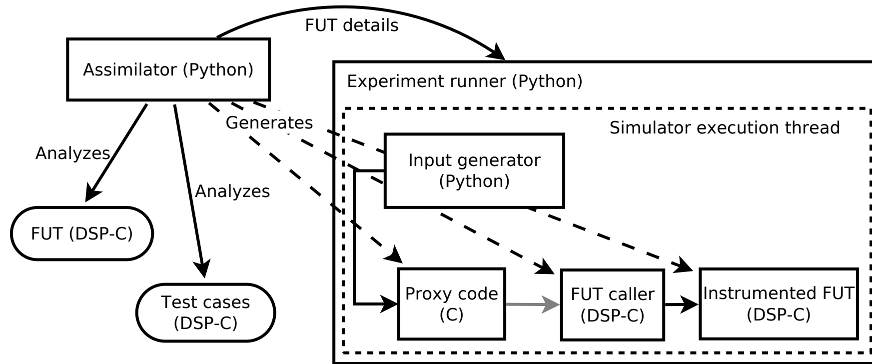

Fig. 1. Prototype's architecture and the execution environment.

least one test function are used to construct the input space, as it is likely that variables and fields which are never set are uninteresting for testing purposes. In particular, this is true for variables which do not affect the execution of the FUT (e.g., output variables), or for variables which are initialized in the setup phase and should not be changed due to the logic of the system.

2.2 Execution Environment

The DSP simulator supports interacting with simulated code in a restricted manner, e.g., direct function calls are not possible. Therefore our implementation was heavily adapted to the simulator.

The `Input generator` (Fig. 1) implements the search algorithms using a vector representation of basic types, which is oblivious to the actual input structures. The `Proxy code` and `FUT caller` (automatically synthesized by the `Assimilator` module) packs the input vector into the input data structures, which is then passed to the simulated DSP-C environment. After each FUT execution observation data is read to calculate the fitness value of the last input vector. Then the execution cycle is repeated with the next input vector generated by the search algorithm.

3 Experimental Evaluation

In this section we compare the experimental results of hill climbing (HC), (1+1)EA, and random search (RS) algorithms for branch coverage on a set of four functions in a library sub-module of Ericsson's middleware code base. Both HC and (1+1)EA algorithms use the same popular fitness function for branch coverage that combines *approach level* and *branch distance* [5].

The particular sub-module was selected as it was considered suitable to be tested using the simulator, rather than the real hardware. In particular, suitable sub-modules should have no or minimal inter-module dependencies, and should be

without timing critical requirements. In this sub-module, functions with the most lines of code and branching statements were selected. The chosen FUTs are listed in Table 1 with structural information. `Func_A`, `Func_B`, and `Func_C` are directly tested by the existing unit tests. However, `Func_subA` is not executed directly but through `Func_A`, which we mimic in our test data generation approach. Therefore, `Func_A` and `Func_subA` share the same input vector.

Table 1. Structural information of the functions that are included in our study.

Function under test	LOC	Number of branches	Input vector size	
			Original	Reduced
`Func_A`	191	20	508	30
`Func_subA`	37	6	508	30
`Func_B`	352	20	143	28
`Func_C`	179	32	659	146

Experiments were run on Ericsson servers using the simulator. We use execution count (i.e., the number of calls to the FUT) for comparison, instead of the clock time, so that the efficiency of our implementation, interactions with the simulator, or the server load do not affect the results. For practical purposes we imposed a limit of 1000 executions (FUT calls) per targeted branch. In order to account for the stochastic nature of the search algorithms, we repeated each experiment 50 times per algorithm per FUT. The total time for executing all experiments was around 150 hours, or 6 days.

Statistical significance tests and effect size measurements are applied for pairwise comparison of algorithms, with respect to branch coverage and execution counts. We omit this analysis for the sake of brevity, which can be found in the technical report [1]. The minimum, maximum, mean, and median values for the achieved branch coverage by each algorithm are shown in Table 2. Branch coverage of the existing test cases are also listed for comparison. For `Func_A` and `Func_subA` all algorithms were able to reach full coverage at most of the runs, while some branches of `Func_B` were covered only by the HC algorithm.

Table 3 shows the number of executions needed per algorithm for each FUT. Again the minimum, mean, median, and maximum values among the 50 independent runs are reported in the table.

The results indicate that both HC, (1+1)EA, and RS were able to achieve high branch coverage most of the time (Table 2). At least one of the algorithms were able to achieve 100 % branch coverage in the case of three out of the four FUTs. For two of the FUTs (`Func_subA` and `Func_B`) the achieved branch coverage was higher than branch coverage of the existing test cases. Therefore, we were able to increase the branch coverage for the FUTs that are studied in our case study. However, search algorithms did not always reach the highest possible branch coverage (`Func_C` in Table 2).

We observed that RS was not much worse than other algorithms at branch coverage, except for `Func_B` where HC was the clear winner. This indicates that

Table 2. Branch coverage achieved by hill climbing, (1+1)EA, and random search algorithms, as well as the existing test cases.

FUT	Exist.	Hill Climbing				(1+1)EA				Random Search			
		min	mean	med.	max	min	mean	med.	max	min	mean	med.	max
`Func_A`	1.0	0.95	0.996	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
`Func_subA`	0.83	0.5	0.99	1.0	1.0	0.5	0.983	1.0	1.0	0.5	0.947	1.0	1.0
`Func_B`	0.85	0.95	0.998	1.0	1.0	0.7	0.701	0.70	0.75	0.7	0.705	0.70	0.75
`Func_C`	1.0	0.656	0.746	0.75	0.812	0.75	0.812	0.812	0.875	0.75	0.824	0.812	0.906

Table 3. Number of executions (fitness evaluations or FUT calls) that each search algorithm needed before terminating.

FUT	Hill Climbing				(1+1)EA				Random Search			
	min	mean	med.	max	min	mean	med.	max	min	mean	med.	max
`Func_A`	330	641	576	1555	288	551	506	1059	73	151	150	310
`Func_subA`	167	448	382	3234	110	493	408	3143	52	436	131	3034
`Func_B`	1853	2231	2194	2838	5650	6401	6364	6955	5382	5996	6042	6070
`Func_C`	10043	11722	11551	13763	7710	9396	9256	11970	5605	7353	7591	9686

many of the branches are relatively easy to cover. One such branch predicate that we found to be common in the code base is inequality comparison with a signed input value and a small constant, such as $(x \leq 10)$. For example, a signed 32-bit integer input x would mean that the probability of a random input leading to the false and true branches approximately equal ($P(false) \approx P(true) \approx 0.5$). Similar situation can be observed if the variable being compared is unsigned type, but is defined as a bit field with, e.g., 4-bit length. In embedded software bit fields are commonly used, in order to reduce the memory usage as much as possible.

Furthermore, on average RS is faster (i.e., needs less number of FUT calls) to cover a branch, if it can, compared to other algorithms (Table 3). It is known in the literature that RS is typically faster at covering easy branches than more informed search algorithms.

3.1 Threats to Validity

Due to practical reasons (such as computational resources) we imposed arbitrary limits on the execution of the algorithms on each targeted branch. Different execution limits might have led to differing results.

We selected limited number of FUTs from one sub-module. FUTs were selected among the functions with more lines of code and number of branches. We do not know if many smaller FUTs, instead of few big ones, would give significantly different results. In the future, we plan to extend this work to more sub-modules and functions in the code base.

4 Discussion and Conclusion

In this paper, we discussed a case study on application of SBST techniques for an industrial embedded software. The implemented tool was heavily adapted to the simulated execution environment (Section 2). This can be seen as a special version of the *execution environment problem*, which usually refers to the concerns about interacting with the file system, the network, or a database in the context of non-embedded systems [4]. Moreover, we did not have enough detailed technical knowledge of the system under test to understand the input space. To overcome this problem, we used existing test cases to automatically craft a test case template (Section 2.1).

During the experiments we were able to increase the total branch coverage of existing test cases. So we can say that the employed SBST techniques indicate beneficial results. However, we observed that randomly generated inputs were as effective as more informed search algorithms in many (but not all) cases, which indicate that there are many branches that are easy to cover. A practitioner may prefer to start with random search to cover easy branches first, before using informed search algorithms to target other branches.

For future work, we would like to extend the case study to more FUTs in a similar code base. We plan to investigate alternative ways of interacting with the simulator to reduce the execution times, which was a practical bottleneck for the experiments. Another interesting idea is the use of hybrid search algorithms that can cover easy branches swiftly (e.g., by using random search) and then move to other branches.

Acknowledgments This work was supported by VINNOVA grant 2011-01377, and the Knowledge Foundation (KKS), Sweden. We would like to thank Jonas Allander, Catrin Granbom, John Nilsson, and Andreas Ermedahl at Ericsson AB for their help in enabling the case study.

References

1. Doganay, K., Eldh, S., Afzal, W., Bohlin, M.: Search-based testing for embedded telecommunication software with complex input structures: An industrial case study. Tech. Rep. 5692, SICS Swedish ICT (July 2014)
2. Leary, K., Waddington, W.: DSP/C: a standard high level language for DSP and numeric processing. In: International Conference on Acoustics, Speech, and Signal Processing, 1990. ICASSP-90. pp. 1065–1068 vol.2 (1990)
3. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)
4. McMinn, P.: Search-based software testing: Past, present and future. p. 153–163. ICSTW '11, IEEE Computer Society, Washington, DC, USA (2011)
5. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841 – 854 (2001)
6. Yang, B., Hu, H., Jia, L.: A study of uncertainty in software cost and its impact on optimal software release time. IEEE Transactions on Software Engineering 34(6), 813–825 (2008)