KALI: Multi-thread Combinatorial Test Generation with SMT solvers

Andrea Bombarda
Department of Engineering
University of Bergamo
Bergamo, Italy
andrea.bombarda@unibg.it

Angelo Gargantini
Department of Engineering
University of Bergamo
Bergamo, Italy
angelo.gargantini@unibg.it

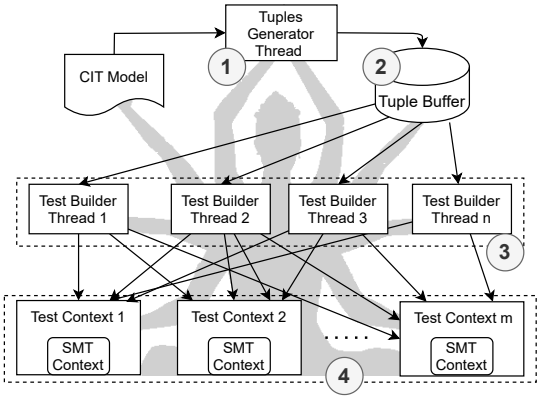

Fig. 1: Data flow for the KALI tool

I. Tool Description

With this document, we submit to the 2nd edition of the CT-Competition KALI [2], a Java tool exploiting an SMT solver, through the library `java-smt`[1], to generate test suites with the desired combinatorial coverage. It implements multi-threading strategies for reducing the test generation time. Moreover, being based on an SMT solver, KALI is able to deal with all the models and tracks of the CT-Competition.

The tool source code, together with other documents and tests is available at https://github.com/fmselab/ct-tools/tree/main/KALI.

A. Tool architecture and algorithm

The data flow we devised for KALI to generate all the tests is shown in Figure 1. All the tuples to be covered are generated by a single thread (step 1 in Fig. 1) from a CIT model in the CTWedge format [3]. The generated tuples are stored into a shared buffer (step 2 in Fig. 1) with limited capacity (40 tuples in our experiments, but users may configure it with a different capacity as preferred). Then, when the shared buffer is fully filled, the tuple generation thread stops and waits until a new free slot is available. This process allows avoiding storing all tuples at the same time, and thus, guarantees a consistent saving in memory utilization, especially for complex combinatorial models, in which the number of tuples can be significantly high.

[1]https://github.com/sosy-lab/java-smt

Algorithm 1 Tuple consumption procedure

Require: $TupBuffer$, the buffer containing the tuple already produced and ready to be consumed
Require: TC, the list of all the test contexts
Require: M_C, the CIT model

▷ Extract the tuple from the tuple buffer
1: $tp_i \leftarrow TupBuffer.extractFirst()$
▷ Try to find a test context which implies the tuple
2: $tc_j \leftarrow findImplies(TC, tp_i)$
3: **if** tc_j is not $NULL$ **then**
4: **return**
5: **end if**
▷ Try to find a test context which is compatible with tp_i
6: $tc_j \leftarrow findCompatible(TC, tp_i)$
7: **if** tc_j is not $NULL$ **then**
8: $tc_j.updateTC(tp_i)$
9: **return**
10: **end if**
▷ Create a new empty test context
11: $tc_j \leftarrow createTestContext(M_C)$
12: **if** $tc_j.isCompatible(tp_i)$ **then**
13: $tc_j.updateTC(tp_i)$
14: **else**
15: $tp_i.setUncoverable()$
16: **end if**
17: **return**

At step 3 in Figure 1, n threads responsible for building tests start to work in parallel and consume tuples from the tuple buffer. The number n can be determined automatically by the tool based on the hardware architecture, as was done in the experiments described in this document, or set by the user. Each thread is capable of taking a tuple tp_i and attempting to add it to any of the available *test contexts*, which are continually updated and generated as needed (as described in step 4 of Figure 1). This process is described in Algorithm 1 and is repeated by each thread until all the tuples have been consumed. In particular, given a tuple tp_i (extracted from the buffer in line 1):

- `findImplies` at line 2 finds, in the list of all test contexts TC, the first test context tc_j, which already implies the tuple tp_i, if there exists one. Then, if tc_j

TABLE I: Average of the results obtained in our experiments with KALI

Model	Size	Time $[s]$	Model	Size	Time $[s]$
UNIFORM_BOOLEAN_0	32.0	0.48	UNIFORM_ALL_0	100.7	1.00
UNIFORM_BOOLEAN_1	32.0	0.48	UNIFORM_ALL_1	282.0	4.95
UNIFORM_BOOLEAN_2	31.7	0.47	UNIFORM_ALL_2	148.0	2.72
UNIFORM_BOOLEAN_3	32.7	0.48	UNIFORM_ALL_3	356.0	4.59
UNIFORM_BOOLEAN_4	28.7	0.44	UNIFORM_ALL_4	30.0	0.48
MCA_0	218.0	2.59	BOOLC_0	35.0	0.65
MCA_1	89.3	0.83	BOOLC_1	61.0	0.71
MCA_2	178.0	1.32	BOOLC_2	54.0	0.65
MCA_3	315.0	3.30	BOOLC_3	16.3	0.49
MCA_4	291.3	2.94	BOOLC_4	4.0	0.48
MCAC_0	2.3	0.81	NUMC_0	437.0	4.58
MCAC_1	39.0	2.16	NUMC_1	440.0	4.12
MCAC_2	7.0	0.81	NUMC_2	180.0	7.36
MCAC_3	119.3	1.53	NUMC_3	91.7	2.33
MCAC_4	24.7	2.10	NUMC_4	81.3	4.87
FM_0	26.7	1.06	CNF_0	26.3	1.89
FM_1	28.7	0.60	CNF_1	163.7	1.72
FM_2	70.7	1.20	CNF_2	210.2	1.97
FM_3	67.0	1.22	CNF_3	361.0	5.11
FM_4	9.3	0.57	CNF_4	304.0	11.84
INDUSTRIAL_0	57.0	0.97	HIGHLY_CONSTRAINED_0	20.3	1.60
INDUSTRIAL_1	277.0	5.62	HIGHLY_CONSTRAINED_1	235.7	3.54
INDUSTRIAL_2	42.7	1.21	HIGHLY_CONSTRAINED_2	22.3	0.70
INDUSTRIAL_3	46.7	1.02	HIGHLY_CONSTRAINED_3	56.0	1.57
INDUSTRIAL_4	18.7	0.68	HIGHLY_CONSTRAINED_4	58.0	1.82

has been found, tp_i is consumed;

- `findCompatible` at line 6 finds, in the list of all test contexts TC, the first test context tc_j that is compatible with the tuple tp_i (i.e., the tuple does not clash with the assignments already performed by the test context and with the constraints of the combinatorial model), if there exists one. Then, if tc_j is found, tp_i is passed to tc_j, which updates its SMT context, and tp_i is consumed;
- if a thread cannot find a test context tc_j in which the tuple tp_i is compatible or implied, a new test context is created. It is initialized by the function `createTestContext` (line 11) which builds a new test context tc_j, together with the corresponding SMT context, and adds all the variables and constraints of the combinatorial problem to tc_j. If tp_i is compatible with the newly created test context, the tuple is consumed, added to the context; otherwise, it means that the tuple is not compatible with the constraints, it is considered as uncoverable and skipped, and the empty test context is discarded.

When all tuples are consumed, each test context provides the resulting test case, which can be complete (if all parameters have been assigned) or incomplete. In both cases, the test case is a model that satisfies the SMT solver context.

II. KALI PERFORMANCE ON EXAMPLE BENCHMARKS

In this section, we report the results obtained with KALI on the example benchmarks[2] given by the organizers of the second edition of the CT-Competition [1] at IWCT 2023 on a machine using a Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (16 physical cores, 32 logical cores) with 256 GB RAM. As per the CT-Competition rules, we have set a timeout of 300 seconds. Moreover, the experiments have been executed 3 times for each model, and the average of the size and generation time is reported in Tab. I.

[2]https://github.com/fmselab/CIT_Benchmark_Generator/tree/main/Benchmarks_CITCompetition_2023

III. CONCLUSION

With this document, we have briefly introduced the tool KALI, previously presented in [2] and we have shown its applicability to the benchmarks given by the CT-Competition organizers. KALI has shown to be able to deal with all the example benchmarks, and to always produce test suites without having timeouts.

REFERENCES

[1] A. Bombarda, E. Crippa, and A. Gargantini. An environment for benchmarking combinatorial test suite generators. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, apr 2021.

[2] A. Bombarda, A. Gargantini, and A. Calvagna. Multi-thread combinatorial test generation with smt solvers. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[3] A. Gargantini and M. Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317, April 2018.