Coverage Metrics for Verification of Concurrent SystemC Designs Using Mutation Testing

Alper Sen
Department of Computer Engineering
Bogazici University
Istanbul, Turkey
Email: alper.sen@boun.edu.tr

Magdy S. Abadir
Design Technologies
Freescale Semiconductor Inc.
Austin, Texas
Email: m.abadir@freescale.com

***Abstract*—Design verification has grown to dominate the cost of electronic system design; however, designs continue to be released with latent bugs. A verification test suite developed for a sequential program is not adequate for a concurrent program. A major problem with design verification of concurrent systems is the lack of good coverage metrics. Coverage metrics are heuristic measures of the exhaustiveness of a test suite. High coverage, in general, implies fewer bugs. SystemC is the most popular concurrent system level modeling language used for designing SoCs in the industry. We propose to attack the verification quality problem for concurrent SystemC programs by developing novel mutation testing based coverage metrics. Mutation testing has successfully been applied in software testing and RTL designs. In this paper, we develop a comprehensive set of mutation operators for concurrency constructs in SystemC. Our approach is also unique in that we define a novel concurrent coverage metric considering multiple execution schedules that a concurrent program can generate. This metric allows us to adequately measure the coverage for concurrent programs. We performed experiments with various designs including a large industrial design and obtained favorable results on multiple applications.**

I. Introduction

Concurrent programs are harder to verify than their sequential counterparts due to their inherent non-determinism and concurrency problems such as interference and deadlock. Raising the level of abstraction via system level design is one of the most efficient methods of reducing complexity due to concurrent electronic designs. It is easier to diagnose concurrency and protocol problems at the system level, whereas these problems are hidden at the lower implementation levels.

SystemC is the most popular system level modeling language used for designing SoCs in the industry. It is a C++ library that contains constructs related to concurrency, time and hardware data types. It is freely available from OSCI [15] and is an IEEE standard. SystemC Transaction Level Modeling (TLM) 2.0 standard is a modeling level of the SystemC language providing concurrent communication structures for complex data types such as transactions between IPs. TLM allows us to describe the design at a much more abstract level than RTL or cycle accurate models and brings several of the benefits of working at a higher level of abstraction. However, system level design coupled with multi-processors is a challenge to EDA tool providers, and system level tools are still in their infancy. New system level techniques of analysis are required to improve reliability of concurrent software running on concurrent hardware.

Even if design verification is successfully completed, there is still a doubt whether specifications (or tests) are comprehensive and if they cover all possible behaviors of the system. Increasingly, there is a need to measure the quality of verification effort. Coverage metrics play an important role in evaluating the confidence in the verification results. In this paper, we develop mutation testing based coverage metrics in order to increase verification test coverage for concurrent system level designs.

Mutation testing assumes a given fault model. We develop a fault model for concurrent SystemC designs to inject functional faults similar to the successful stuck-at fault model for manufacturing faults. We show the effectiveness of this fault model by relating the faults to actual bug patterns. Then we generate mutations based on our fault model and insert these mutations into a given SystemC TLM 2.0 design (obtaining a mutant). Finally, we simulate the mutant with the test suite, activating and propagating the mutation to the outputs. In case the outputs of the mutant and the original design are different for a given test, we say that the test kills (detects) the mutant. The higher the percentage of killed mutants, the higher the confidence in the verification test suite.

Our work differs from sequential program based mutation testing. When testing a sequential program, one can assume that there exists only one correct output for a given verification test. This is not true for concurrent programs. Concurrent execution of several processes may result in different schedules and outputs, depending on the order of execution of the different processes chosen by the scheduler. A verification test suite developed for a sequential program is not adequate for a concurrent program. We need concurrency aware verification test suites for concurrent programs. To help develop and measure the quality of such test suites we need a concurrency aware coverage metric. In this work, we develop a novel concurrency aware mutation coverage metric where all possible execution schedules of concurrent programs are considered.

We performed experiments with various designs including a large industrial design to validate the effectiveness of our techniques. Our experimental results confirm the inadequacy of current verification tests for checking concurrent features of SystemC. We summarize the contributions of this paper as follows.

- We present a comprehensive list of mutation operators for concurrent SystemC and show the effectiveness of these operators by relating them to actual bug patterns.
- We develop a novel concurrent mutation coverage metric using multiple schedules of a concurrent program that allows us to adequately measure the coverage for concurrent programs.
- We develop an automated mutation testing based coverage framework and provide experimental results on multiple applications showing the effectiveness of our technique in demonstrating the need for concurrency aware verification test suites.

II. Related Work

There has been work on traditional coverage metrics for RTL and gate level designs. These techniques can be summarized as code coverage, structural coverage, functional coverage, and observability-based coverage techniques [19], [7], [8].

Code coverage techniques measure the amount of activation of lines, branches and expressions in designs source code during simulation. This is a limited approach since activations cannot be observed at the outputs of the design, hence this approach may not have impact on the design. Structural coverage techniques extract an abstract state machine from design descriptions and measure the number of states traversed during simulation. This approach is limited due to the growing size of the state machine for complex designs. Functional coverage metrics target design functionalities to cover interesting scenarios. However, these techniques are not automatic. They need to be redeveloped for new designs since they rely on internal monitors defined and built by engineers having knowledge both of the design specification and implementation. Observability-based coverage metrics observe the impact of errors activated by the verification tests at the outputs of the design. All above techniques have been applied at the lower level designs such as RTL and gate level designs.

Mutation testing technique is an observability-based coverage technique based on software testing [6], [13], [14]. It has been applied to programming languages such as Java [5], [12] and state machines [16]. Mutation testing is based on a given fault model. Mutations developed based on this fault model are injected into the design one at a time, and it is checked whether such mutations are activated and propagated to the outputs of the design. Mutation testing helps to strengthen the quality of verification tests iteratively until reaching a given target coverage. The previously developed stuck-at fault model is similar to mutation testing and has been very successful for manufacturing faults [1].

Typical mutation operators for a programming language are designed to modify variables and expressions by replacement, insertion or deletion operators. For example, a simple non-concurrent arithmetic operation mutation can change an assignment statement like $x = y + z$ into $x = y - z$, $x = y * z$, or $x = y/z$.

Mutation operators have been defined for concurrency constructs in Java [5], [9]. We make use of some of these operators in our work and enhance them for SystemC. Mutation testing has recently been applied to Verilog [10] and SystemC TLM 2.0 communication interfaces [3], [4]. Our work differs from these works in that we are concerned about all concurrency constructs in SystemC rather than only communication constructs. SystemC and TLM libraries have been modified in those works. We do not modify any libraries, hence we can provide better integration into already developed industrial frameworks. Furthermore, our mutation coverage criterion is different from the earlier ones.

Multiple schedules of a SystemC design have been used for verification purposes before. We earlier used multiple schedules for assertion based verification of SystemC [18]. Helmstetter et al. [11] generate non-equivalent (no two schedules have the same output) SystemC execution schedules using dynamic partial order reductions. This allows them to explore all possible schedules with the same test suite. Our work is different in that we provide a coverage metric to determine the quality of the test suite, that is, for each schedule, we check whether the test suite can detect inserted mutations.

III. Concurrency in SystemC

SystemC has the concept of processes to model the concurrent activities of a system. Processes can be combined into modules to create hierarchies. Processes access external channel interface through the ports of a module and can run concurrently, but code inside a process is sequential. There are two types of processes, method process, and thread process.

The SystemC scheduler controls the timing and order of process execution, handles event notifications and manages updates to channels. It is an event-based simulator similar to Verilog. SystemC processes are non-preemptive. Hence, a process has to voluntarily yield control for another process to be executed.

Processes are triggered and synchronized with respect to sensitivity on events. There are two types of sensitivity. Static sensitivity is defined before simulation starts such as sensitivity to a clock signal, and dynamic sensitivity is defined after simulation starts and can be altered during simulation. Events are controlled via $wait$, $notify$ and $next\_trigger$ functions of the sc_event class. A $wait$ function changes dynamic sensitivity of a thread process and suspends its execution. For example, $wait(0)$ delays the process by one delta cycle, whereas the process waits on event e with $wait(e)$. Similarly, a $next\_trigger$ function changes dynamic sensitivity of a method process. However, this function returns immediately rather than suspending execution. Events occur explicitly by using the $notify$ function and the scheduler resumes execution of a thread or method process by executing the internal $trigger$ function. For example, $e.notify()$ is called an immediate notification since processes sensitive to event e will run in the current evaluation phase or delta cycle.

```
initially cs1 = true;                    initially cs2 = false;
T1:                                      T2:
   e.notify();                              wait(e);
   cs1 = false;                             cs2 = true;
```

Fig. 1. SystemC Example

Figure 1 displays a simple SystemC example where there are multiple schedules during the execution of a design. In this example, thread T2 is waiting for an event from thread T1 to move forward. $cs1$ and $cs2$ are outputs of the design and are initially $true(t)$, $false(f)$, respectively. There are two possible execution schedules by the kernel. These are T1;T2 and T2;T1;T2. If T1 is executed before T2 (T1;T2), then $notify$ message is lost since T2 is not waiting and this leads to a special case of deadlock for T2. In the other schedule (T2;T1;T2), both threads run till completion.

Similar to events; channels, interfaces, and ports are used for process synchronization and also for communication refinement. These constructs are the core of Transaction Level Model (TLM) based methodology and are defined as follows. Channels hold and transmit data. A channel implements interfaces. An interface is a declaration of the available functions for accessing a given channel and describes the set or subset of operations that the channel provides. A port is bound to a channel through an interface, and it is an agent that forwards function calls up to the channel on behalf of the calling module. Channel functions $read$, $write$, $b\_transport$, $nb\_transport\_fw$, $nb\_transport\_bw$, put, get, $peek$, nb_put, nb_get, nb_peek generate synchronization between processes.

Synchronization is also established through instantiating $sc\_semaphore$ and sc_mutex objects, which provide $wait$, $trywait$, $post$ and $lock$, $trylock$, $unlock$ functions, respectively. Table I summarizes SystemC concurrency functions.

TABLE I
SYSTEMC CONCURRENCY FUNCTIONS

Construct	Available Functions
Event	$notify$, $wait$, $next\_trigger$
Channel	$read$, $write$, put, get, $peek$, nb_put, nb_get, nb_peek, $b\_transport$, $nb\_transport\_fw$, $nb\_transport\_bw$
Semaphore	$wait$, $trywait$, $post$
Mutex	$lock$, $trylock$, $unlock$

IV. SEQUENTIAL AND CONCURRENT COVERAGE METRICS

A verification test suite developed for a sequential program is not adequate for a concurrent program. We need concurrency aware verification test suites for concurrent programs, and to help develop and measure the quality of such test suites, we need a concurrent coverage metric.

In this section, we will define a new concurrent mutation coverage metric for concurrent systems with multiple schedules. First, we define a $mutant\ P'$ as the introduction of a mutation into a program P. Next, we modify the traditional definition of "killing (detecting) mutants" given for a single schedule in mutation testing to account for multiple schedules.

Definition 1: (killing mutant for a single schedule) Given a mutant P' for a program P and a test t, t is said to $kill\ P'$ if and only if the output of t on P' is different from the output of t on P.

Definition 2: (killing mutant for multiple schedules) Given a mutant P' for a program P and a test t, t is said to $kill\ P'$ if and only if there exists an output of t on P' that is different from all possible outputs of t on P.

Note that the latter definition emphasizes the multiple schedules that can be generated by a concurrent program. A verification test is expected to kill each mutant with at least one test case. In case a mutant cannot be killed, the tester needs to show that 1) (for multiple schedules): outputs of t on P' are contained in outputs of t on P; (for a single schedule): output of t on P' is the same as the output of t on P or, 2) update tests by adding a test case that kills the mutant.

We now define *Mutation Coverage* to determine the coverage of verification tests using Definition 1 and Definition 2.

Definition 3: (sequential mutation coverage) The ratio of the number of mutants killed using Definition 1 to the number of all mutants is called a sequential mutation coverage for a single schedule.

Definition 4: (concurrent mutation coverage) The ratio of the number of mutants killed using Definition 2 to the number of all mutants is called a concurrent mutation coverage for multiple schedules.

We illustrate the need to have a concurrent coverage metric and how it can improve test suite quality with a mutant obtained from the example in Figure 1. We apply a mutation operator that removes the concurrency construct $wait$ and obtain the mutant in Figure 2. Assume that the test suite only contained Test1, which generates only the final values of $cs1, cs2$ as can be seen from Test1 columns in Table II. We denote values that are generated before the thread ends or deadlocks as final values. If we considered only a single schedule (T1;T2) of the program, this mutation is considered killed if Definition 1 is used because output of the original program (ff) differs from the output of the mutant (ft). However, the mutation is not killed if Definition 2 is used, since outputs of both schedules of the mutant are the same (ft), and this output is one of the possible outputs of the original program. Hence, we can falsely obtain higher mutation coverage if we do not use a concurrent metric. Since the coverage is low using concurrent metric, we need to improve it by adding a new test. One possible way to improve the quality of the test suite (and kill the mutant) is to generate values of $cs1, cs2$ at

```
initially cs1 = true;                          initially cs2 = false;
T1:                                            T2:
   e.notify();                                    // removed wait(e)
   cs1 = false;                                   cs2 = true;
```

Fig. 2. SystemC Mutant Example

all value changes, starting from their initial values tf. We call this test as Test2. From Table II, it is clear that the mutant output tf, tt, ft is not contained in the outputs of the original program, hence the mutant is killed. Another possible way to improve the quality of the test suite would be to use an assertion to check whether $cs1, cs2$ can be true at the same time. Note that the second schedule (T2;T1;T2) is exhibited as (T2;T1) in the mutant.

TABLE II
CONCURRENT AND SEQUENTIAL COVERAGE METRIC COMPARISON

	Test1 Outputs(cs1cs2)		Test2 Outputs(cs1cs2)	
Schedule	Original	Mutant	Original	Mutant
T1;T2	ff	ft	tf,ff	tf,ff,ft
T2;T1;T2	ft	ft	tf,ff,ft	tf,tt,ft

V. MUTATION OPERATORS FOR CONCURRENT SYSTEMC

In this section, we first identify typical bugs that designers make when using concurrency. Next, we develop mutation operators for concurrent functions in SystemC and relate them to these bugs.

The following list of bug patterns is based on resources such as Java concurrency bug patterns [5], [9], patterns for TLM 2.0 communication functions [3], and our experience.

B1. Lost notify: If a notify() is executed before its corresponding wait(), the notify() has no effect and is lost. As a result, code executing a wait() might not be awakened because it is waiting for a notify() that occurred before the wait() was executed.

B2. Interference: Two or more concurrent threads access a shared variable and at least one access is a write.

B3. Deadlock: Two or more processes are unable to proceed due to waiting for one another. Also, we say that a process is deadlocked when it is stuck as in Figure 1.

B4. Starvation: A process may starve due to actions of other processes. A change in lock acquisition may lead to this.

B5. Resource exhaustion: A group of processes hold all of a finite number of resources. One of them needs additional resource but no other process gives up.

B6. Incorrect count initialization: This occurs when the number of entries in a semaphore initialization is incorrect.

B7. Nondeterminism: If an immediate notification is used this may cause nondeterminism. Process execution can be interleaved with immediate notification, and the order of runnable processes are executed is undefined.

B8. Forgetting functions: Forgetting to call a put before a get function.

B9. Incorrect functions: Using $read$ instead of $write$, or using blocking instead of a nonblocking function.

We now present a set of mutation operators designed to exercise concurrency and synchronization present in SystemC programs. We describe operators in two categories. In the first category, mutations modify parameters of concurrency functions. In the second category, mutations remove, replace, or exchange concurrency functions. We also relate these mutation operators to real bug patterns described above.

Category 1: Modify parameters of concurrency function.

M1. Modify Function Timeout: This operator can be applied to functions with a timeout parameter such as $wait$, $notify$, and $next\_trigger$. For example, we can modify $wait(time)$ to $wait(time/2)$. This modification may result in an interference or data race bug B2 since a process may access a shared variable when it is not supposed to, due to a potential change in process activation time or order. This mutation can also lead to a lost notify B1 and deadlock B3 if the notification is sent before the corresponding wait.

M2. Modify Concurrency Construct Count: This operator can be applied to semaphores that indicate the number of threads that can access a shared resource. This mutation leads to incorrect count initialization B6 and may also lead to resource exhaustion B5 if the count is decremented. For example, modify $sc\_semaphore(num)$ to $sc\_semaphore(num-1)$ or $sc\_semaphore(num+1)$.

Category 2: Remove, replace, exchange concurrency function.

M3. Remove Concurrency Construct: This operator removes calls to concurrency functions described in Table I. This mutation results in B8 bug. Removing $wait$ or $lock$ may result in interference B2 since a thread that should have been waiting can potentially access a shared variable. Removing $notify$ may result in lost notify bug B1 and deadlock B3. Removing an $unlock$ may result in the process waiting for the lock to be starved, hence B4.

M4. Replace Timed Construct with Untimed Construct: This operator replaces timed construct with an untimed construct and vice versa. For example, we can replace timed notify with immediate notify which may result in nondeterminism B7. Also, a $wait(e)$ can be replaced by $wait(1, SC\_NS)$ which may result in interference B2 since the process may access a shared variable before waiting for the appropriate event notification. Also, this can result in an incorrect function B9.

M5. Exchange Lock/Permit Acquisition: This operator exchanges a function of a semaphore or mutex with

TABLE III
RELATING BUG PATTERNS AND MUTATION OPERATORS

Concurrency Bug Pattern	Mutation Operators
B1. Lost notify	M1, M3, M6
B2. Interference	M1, M3, M4, M6, M7
B3. Deadlock	M1, M3, M6, M7
B4. Starvation	M5, M6, M3
B5. Exhaustion	M2, M6
B6. Incorrect count	M2
B7. Nondeterminism	M4
B8. Forgetting functions	M3
B9. Incorrect functions	M4, M5, M6, M7

another one. In a semaphore, $wait$ or $trywait$ can be used to acquire permits to a shared resource. Exchanging one function with another may lead to timing changes and starvation. For example, using $trywait$ instead of $wait$ may lead to starvation B4 since in the first case threads do not block whereas threads block in the second case. Also, this can result in an incorrect function B9.

M6. Exchange Function Call with Another: This operator exchanges a function in Table I with another appropriate function. For example, a call to semaphore $post$ is exchanged with $wait$ or $notify$ exchanged with $wait$. Also, a call to get may be exchanged with appropriate $peek$. This may result in starvation B4 as in M5, interference B2 since a shared variable may be accessed, deadlock B3 since instead of releasing a lock it may be requested creating a circular chain of lock requests, exhaustion B5 since resources may have been received from the channel by get, lost notify B1, or incorrect function B9.

M7. Exchange one Concurrency Construct Instance with Another: When there is more than one lock (mutex or semaphore), we replace a call to a lock with another one. This may lead to a deadlock situation B3, or interference B2 since the correct lock is not used to access critical regions, and also may lead to an incorrect usage of functions B9.

We have described mutation operators and how they relate to real bug patterns. Table III summarizes this relationship. Our mutation operators are equally applicable for a single schedule or for multiple schedules of a concurrent program.

VI. MUTATION COVERAGE ALGORITHM

Algorithm MutCov in Figure 3 displays our automatic mutation coverage algorithm for SystemC. In line 1, for every concurrency function in the original program, we insert a mutation operator and obtain a $metaprogram$. We currently use an implementation with parsers to accomplish this. The metaprogram uses a mutant schema where each inserted mutation is guarded by a conditional statement that can be switched on and off at runtime. This metaprogram is more efficient than generating a new version of the program for each mutation. Note that not all mutation operators are applicable to every program. In line 3, we enable the inserted mutation operators one by one, obtaining a mutant for each enabling. Then, in line 4, we choose every test in the test suite one by one for simulation. It is possible that some tests may not execute the inserted mutation so we do not consider these tests during simulation. This check can be done by an initial execution of all tests on the metaprogram. In line 5, we use the work by Helmstetter et al. [11] for simulation. During simulation, there can be multiple execution schedules, and we may obtain multiple outputs, that is, one output for each schedule. In line 6, if there is a single output of the simulation, we check if the test kills the mutant using Definition 1, otherwise, we use Definition 2. Finally, in line 9, we generate mutation coverage using Definition 3 and Definition 4.

```
Algorithm MutCov
Input: a SystemC program $P$, a set of verification tests $T$
Output: sequential and concurrent mutation coverage

1:  insert all "relevant" mutation operators into $P$;
    (all mutation operators may not be applicable)
2:  for each inserted mutation operator $M$ in $P$ do
3:      enable $M$ and obtain a mutant $P'$;
4:      for each verification test $t \in T$ do
5:          simulate $P'$ with $t$;
6:          check if $t$ kills $P'$;
7:      endfor;
8:  endfor;
9:  generate sequential and concurrent mutation coverage
```

Fig. 3. Mutation Coverage Algorithm

VII. EXPERIMENTAL RESULTS

We have performed experiments on concurrent SystemC designs using our mutation testing based coverage framework to validate the effectiveness of our approach. As experimental testbeds, we chose five designs from OSCI SystemC and TLM 2.0 library distributions [15], two designs from SystemC Runtime Verification Toolbox (SCRV) [17], and an industrial design. Each design contained its own verification tests. For each design, we executed Algorithm MutCov and displayed mutation coverage together with the mutations killed and not killed as feedback.

The OSCI and SCRV design descriptions can be found in respective references. The industrial design is a modeling library based on SystemC used for architectural exploration, RTL development, constrained random verification, and early software development. It includes a complete set of BFM and monitor components for several bus protocols including proprietary TLM compliant bus protocols. Also, included is a new testbench environment built on the existing hardware modeling library, that includes controllers through which all interaction with the device under verification takes place. Some of the controllers are data stimulus, clock, signal, bus, fifo, and memory. The framework consists of around 40,000

TABLE IV
MUTATION TESTING BASED COVERAGE METRIC EXPERIMENTS

Design	Lines	# Schedules	# Mutants	# Killed Mutants (single schedule)	# Killed Mutants (multiple schedules)	Coverage (%) (sequential)	Coverage (%) (concurrent)
pv_example	2449	1	18	12	12	66	66
p2p_pipe_thread	926	1	36	20	20	55	55
scx_mutex_w_policy	161	1	30	23	23	76	76
at1_phase	2350	1	34	19	19	56	56
industrial	40,000	1	146	76	76	52	52
pvt_put_example	1161	2	55	23	20	41	36
bozo	139	3	16	9	1	56	6
sirac	253	12	28	17	5	60	18

lines of SystemC code. The experiments were executed with up to 12 threads. We used 17 testbenches that were used during original validation. This design contained only a single schedule.

Table IV shows our results. In the table, we denote the number of lines in the design in column $Lines$, the number of schedules that the original program can generate in column $Schedules$, the number of generated mutants in column $Mutants$. Columns denoted by *Killed Mutants* display the number of killed mutants using Definition 1 and Definition 2. Finally, columns denoted by $Coverage$ represent sequential and concurrent mutation coverage percentages. All original designs except the last three in Table IV have a single schedule. Hence, we obtain the same coverage percentages for both sequential and concurrent coverage. For designs with multiple schedules, we used the first schedule of the original and mutant designs when obtaining sequential coverage. We observed that design $pv\_example$ has two mutants with multiple schedules although the original program has a single schedule.

Our experimental results can be summarized as follows:

1. Low sequential and concurrent mutation coverage percentages (less than 80%) confirm the inadequacy of test suites to find many possible design errors since mutations are closely related to actual errors as we described above.
2. Our experiments also confirm that for designs with multiple schedules, concurrent coverage is smaller than sequential coverage. This makes sense because a concurrent design needs to be tested by a concurrency aware verification test suite that considers multiple schedules. Hence, it is better to use a concurrent coverage metric.
3. For each design (except for the industrial), execution of our mutation algorithm for all mutants took less than 2s to complete except for the last test case, where it took 8.5s since it has the highest number of schedules. This confirms our expectation that multiple schedules will consume more resources than a single schedule. Also, designs with higher number of testbenches result in the larger the execution times (close to 5x execution time slowdown for the industrial design). However, our approach lends itself to simple parallelization and we expect to reduce these execution times substantially when we implement parallelism.
4. The relatively few number of generated mutants shows that by solely focusing on concurrency functions at TLM, we do not suffer from an explosion in the number of mutants. The industrial design especially used very few concurrency constructs.

Similar to other coverage measures, a target coverage percentage can be provided by the user. Empirical industrial data has shown that mutation coverage over 80% could be the initial target [2]. The user can iteratively improve the test suite by adding new test cases until the target coverage is reached. For example, in the case of the industrial design, although the coverage was 52%, we increased it to 83% after the addition of a new test. Our framework can also be used to optimize the test suite by removing redundant test cases that kill the same set of mutants. Ultimately, the quality of the test suite is improved due to our mutation testing based coverage metrics.

VIII. CONCLUSIONS AND FUTURE WORK

There is great demand for developing system level coverage metrics that can adequately deal with concurrency. Mutation testing based coverage metrics are practical, automatic and at the same time allow us to measure the impact of faults on the design. We developed and implemented mutation coverage metrics for concurrent SystemC designs.

The mutation operators in this paper are a comprehensive list of the concurrency and synchronization related features of SystemC. We also showed the effectiveness of these operators by relating them to actual bug patterns.

Our approach is unique in that we defined a novel concurrent mutation coverage metric considering multiple execution schedules that a concurrent program can generate. This metric allows us to adequately generate coverage for concurrent SystemC programs and ultimately, improve the quality of test suites. Our experimental results confirm the inadequacy of current verification test suites for checking concurrent features of SystemC and demonstrate the effectiveness of mutation testing based coverage metrics.

Upon considering all possible schedules of a design the execution times can be high. For future work, in order to improve performance for large designs, we will develop a technique that calculates an approximated set, defined as a subset of all possible schedules of original program. Also, a parallel version of our work will substantially improve performance. Finally, our work is an important step towards standardizing a mutation fault model for SystemC.

Acknowledgments

This research was supported by a Marie Curie European Reintegration Grant within the 7th European Community Framework Programme and BU Research Fund 09HA101P.

References

[1] M. Abramovici, B. M. A., and A. D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, New York, 1990.

[2] B. Bailey. Can Mutation Analysis Help Fix Our Broken Coverage Metrics? In *4th International Haifa Verification Conference*, 2008.

[3] N. Bombieri, F. Fummi, and G. Pravadelli. A Mutation Model for the SystemC TLM 2.0 Communication Interfaces. In *Proc. of the Conference on Design Automation and Test in Europe (DATE)*, pages 396–401. ACM, 2008.

[4] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe. Functional Qualification of TLM Verification. In *Proc. of the Conference on Design Automation and Test in Europe (DATE)*. ACM, 2009.

[5] J. Bradbury, J. Cordy, and J. Dingel. Mutation Operators for Concurrent Java (j2se 5.0). In *Workshop on Mutation Analysis, 2006*, Nov. 2006.

[6] T. A. Budd. Mutation Analysis: Ideas, Examples, Problems and Prospects. In *Computer Program Testing*, pages 129–148. North-Holland, 1981.

[7] S. Devadas, A. Ghosh, and K. Keutzer. An Observability-based Code Coverage Metric for Functional Simulation. In *Proc. of the International Conference on Computer Aided Design (ICCAD)*, pages 418–425. IEEE Computer Society, 1996.

[8] F. Fallah, S. Devadas, and K. Keutzer. OCCOM: Efficient Computation of Observability-based Code Coverage Metrics for Functional Verification. In *Proc. of the Design Automation Conference (DAC)*, pages 152–157. ACM, 1998.

[9] E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How to Test Them. In *Proc. of the International Parallel and Distributed Processing Symposium*, 2003.

[10] M. Hampton and S. Petithomme. Leveraging a Commercial Mutation Analysis Tool for Research. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007*, pages 203–209, Sept. 2007.

[11] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip. In *Proc. of the International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2006.

[12] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: An Automated Class Mutation System: Research Articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.

[13] J. Offutt, P. Ammann, and L. Liu. Mutation Testing implements Grammar-Based Testing. In *Workshop on Mutation Analysis, 2006*, pages 12–12, 2006.

[14] J. Offutt and R. H. Untch. *Mutation 2000: Uniting the Orthogonal*. Kluwer Academic Publishers, 2001.

[15] Open SystemC Initiative, http://www.systemc.org/.

[16] S. Pinto Ferraz Fabbri, M. Delamaro, J. Maldonado, and P. Masiero. Mutation Analysis Testing for Finite State Machines. In *5th International Symposium on Software Reliability Engineering*, Nov 1994.

[17] SystemC Runtime Verification Toolbox (SCRV), http://my-trac.assembla.com/scrv/wiki.

[18] A. Sen, V. Ogale, and M. S. Abadir. Predictive Runtime Verification of Multi-Processor SoCs in SystemC. In *Proc. of the Design Automation Conference (DAC)*, pages 948–953, 2008.

[19] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., 2005.