Abstract— Time-to-Market plays a central role on System-on-a-Chip (SoC) competitiveness and the quality of the final product is a matter of concern as well. As SoCs complexity increases, the validation effort reveal itself as a great barrier, consuming a considerable fraction of the total development time and resources. This work presents a methodology for automatic test vector generation for SystemC designs based on code coverage analysis that is complementary to the functional testing. Instead of create all vectors necessary to guarantee total coverage of the design, it uses code coverage information to generate test vectors to cover the portions of code not exercised by the Black-box Testing. Vectors are generated using a numerical optimization method which does not suffer from restrictions related to symbolic execution such as defining array reference values and loop boundaries. By using the methodology, we expect to guarantee total coverage of the DUV minimizing the fault of omission problem, undetectable by Structural testing.

I. INTRODUCTION

As VLSI and System-on-a-Chip (SoC) technologies advance into the multi-million gates designs, productivity becomes one of the bottlenecks of the design flow. This well known problem, referred as productivity gap [1], has been the motivation for the development of new techniques such as component reuse and platform-based design [1]–[5]. System-level description languages (SDLs) play a central role in this new business model as they make the design cycle more straightforward and the product of each iteration more reliable.

SystemC is an SDL that is becoming a standard for the description of complex SoCs and has been used by major silicon companies like Intel, STMicroelectronics, Philips and Texas Instruments [6]. It enables simulation of hardware and software and guarantees a fast learning-curve, once it is based on C++ open standard. The following characteristics also make it attractive:

1) The same language can be used from the earlier stages of the design cycle to the RTL level;
2) Modules described at different levels of abstraction, and even software, can be simulated together;
3) Designers can explore the performance of different architectures easily, as higher levels of abstraction simulate faster than low-level HDLs;
4) Verification can take place earlier in the design process.

In the complex SoCs context, verification is a major issue, consuming about 70% of the total development effort [7]. Despite the verification effort has been pointed as a problem in studies over the last decade [2], [5], the number of 1-spin successful chips is decreasing [8]. This may indicate that the first generation of verification tools and methodologies were not very well successful in attend the complexity increasing in SoC designs or, at least, they were not largely adopted by the industry.

SystemC was not defined with formal analysis in mind and recent works were intended to allow more efficient verification using it. In [9] it is proposed a methodology based on a combination of static code analysis and SystemC semantics described with abstract state machines (ASM). The authors assume most of SoCs are composed of multiple Intellectual Properties (IPs) which has already been verified individually. Hence, the verification process of the SoC properties are mostly related to transactions which can be represented efficiently as state machines. Once compiled, the finite state machines can be used for formal verification by external tools linked to ASM.

Another approach [10] presents a toolbox for analysys of SoC described in SystemC at the transactional level. The proposed tools extract the formal semantics of the SystemC design and build a set of parallel automata. This intermediate representation can be connected to existing formal verification tools via appropriate encodings, allowing formal verification of the design.

Despite the results achieved by these works, formal verification is not the mainstream methodology in the industrial context and verification will continue to rely on simulation for the foreseeable future [11]. Tools and methodologies should be directed to the automatic generation of the testbench and the input test vectors necessary to efficiently exercise the Design Under Verification (DUV).

In [12] it is proposed an interesting tool for automatic testbench generation. VeriSC generates templates of the testbench’s modules according to the DUV’s characteristics, leaving to the verification engineer to create the signal handshaking, functional coverage metrics and input value distributions. SystemC Verification Library (SVC) classes are created au-
automatically to generate the inputs for the design and for a Reference Model (RM) written in any high-level executable language. The Design and the RM’s output data are then compared to determine whether the DUV behaved correctly.

Several works have also been written about automatic input vectors generation, most of them related to Black-box testing applied to pure software development. In such techniques, there is no need for previous knowledge of the DUV’s internal structure, and the behavior of the DUV is compared against a requirements specification. In [13], it is proposed an automatic test generation using checkpoint encoding and antirandom testing which has achieved better results than pure random test generation [14].

A structural approach for test data generation is presented in [15] where a system to generate test data for ANSI Fortran programs is described. The system executes symbolically a program path in order to create a set constraints to the input variables. From this constraints it is possible to extract the necessary input values to exercise the given path, if the path is feasible. Symbolically executing a program is a computational expensive task, and raise some technical difficulties such as dealing with variable arrays, loop conditions and module calls.

Gallagher et al. used another approach [16]. The problem of test data generation is treated as a numerical optimization problem and the method does not suffer from the technical difficulties associated to symbolic execution systems. Many of Gallagher’s contributions are used in this methodology.

This work presents a methodology for automatic test vector generation for SystemC designs based on code coverage analysis that is complementary to the functional testing. It uses the information provided by a code coverage tool to find a path in a Coverage Flow Graph (TFG) that leads to the uncovered data-flow units. The constraints in this path are used by an optimization method to extract the test vectors necessary to guarantee total coverage of the design.

Concepts of Code Coverage are presented in section II. Section III gives an overall look of the methodology used for automatic test generation. Section IV describes the instrumentation of the source code and section V describes the Numerical Method for Test Generation. An example is given in section VI and concluding remarks are presented in section VII, where it is also pointed the direction for future works.

II. CODE COVERAGE ANALYSIS

Code coverage analysis is a Structural testing technique that compares the program under test behavior against its source code apparent intention. Code coverage analysis tools automate code coverage analysis by measuring coverage [17]. Their results are used as an indirect measure of the quality of the test suite, indicating portions of code not exercised by the tests. The completeness of a test is measured in terms of the fraction of data-flow units (DUFs), such as statements and branches, covered by it. Many definitions for types of code coverage can be found in the literature. This work uses the ones described below [14], [17], [18]:

- **Statement Coverage** - The fraction of blocks exercised by the test data;
- **Branch Coverage** - The fraction of branches evaluated both true and false in a program;
- **Relational Coverage** - The fraction of boundary limits exercised in each relational operator, catching common mistakes like using a “<” where a “<=” is intended;
- **Loop Coverage** - The fraction of loops covered by the tests. Complete loop coverage requires that a loop condition should be executed once, several times and also should be skipped;
- **Path Coverage** - The fraction of execution paths from the program’s entry point to its exit covered by the test.

Some of these coverage types could be infeasible in some programs. For example, the number of paths in a program increases tremendously with each additional loop or branch, making it impossible to achieve 100% path coverage in most of the designs. Additionally, code coverage cannot help finding every possible fault on a given design. The most common fault overlooked by this technique is the fault of omission when some required feature is not implemented. On the other hand, intuition suggests a relationship between test coverage and reliability. Malaiya et al [18] confirmed a positive relation between them and proposed a model that can predict reliability from test coverage measures. From this perspective, a tool capable of generate input vectors based on code coverage can improve the quality of the final product.

III. THE METHODOLOGY

Black-box testing techniques are based on the functional requirements and on random testing in some extension, not considering the internal structure of the DUV. Such an approach can leave portions of code not exercised, such as rarely executed modules or cases which have low controllability. Even using code coverage tools, the task of testing thoroughly a design can be difficult, as such tools only produce source code information which the verification engineers could find difficult to interpret. In cases that automatic code generation or automatic synthesis tools are used, this situation can be more complex and even good programmers would need a lot of time to completely understand the results.

The methodology proposed in this work combines Black-box testing techniques with Structural testing in order to improve test performance. The information regarding portions of code not exercised by the first test cycle (Black-box testing) is used to generate vectors capable of exercise the unverified code during a second cycle (Structural testing). As the Black-box testing is expected to cover most of required features of the DUV, we expect to guarantee total coverage of the design minimizing the fault of omission problem, undetectable by Structural testing alone. The methodology is summarized in figure 1.
Before the functional testing, the DUV source code must be instrumented. Code instrumentation allow a fine control of the program execution and it is explained in detail on section IV. The instrumented source code is compiled to obtain two products: 1) The DUV AST and 2) The DUV Executable Model.

An AST [19] is a condensed form for representing language constructs. In such representation, operators and keywords are associated with the interior nodes and the node leaves represent the operands. The DUV AST would provide us with the program flow information necessary to create a DUV Coverage Flow Graph (TFG).

The TFG represents the program flow through the data-flow units. Its arcs are the necessary conditions to reach a vertex and the vertices represent the data-flow units covered by that condition. Figure 2 shows an example of a Coverage Flow Graph (TFG) construction for branch coverage. Each branch is identified by its line number in the source code.

After code instrumentation and compilation, the Functional Testing Vectors are applied to the DUV Executable Model. Data-flow units covered by the test are recorded. The remaining DFUs are marked as uncovered. For each uncovered DFU a set of paths is traced from the beginning of the TFG to the vertex containing that DFU. Some criteria should be used to choose one among all the possible paths, for example the number of vertices it passes through. The set of constraints of the chosen path represents the necessary conditions to execute its branches and are explained in detail in section V.

A numerical optimization method is used to generate the input values that leads the program flow through the chosen path. The numerical optimization routine is inserted into the design source code together with the instrumentation during the first step. This approach has advantages over symbolic execution as the optimization method has access to all variables, making unnecessary to maintain its values and path conditions as algebraic expressions. Control conditions are placed in every branch to allow forcing the branch execution. The path is executed repeatedly trying to minimize the penalty function of the path constraints. A failure to find the correct input vectors to the path can mean that [16] the path is infeasible, the path may or may not be feasible but the optimization search has become stuck in a local minimum or the initial starting point is too far from the solution.

IV. INSTRUMENTATION

In code instrumentation, each predicate in a branch is substituted by a function call, as presented in figure 3. This step is needed to extract coverage information of the first test cycle and to use the numerical method of test data generation [16].

1: If (c == 1)
2: c = a + b;
3: else
4: c = a - b;

1: If (store(eq, c – 1, c == 1, 1))
2: c = a + b;
3: else
4: c = a - b;

Store is a global function that records information about condition statements. It has been modified from the one presented in [16] and has four parameters:

1) type - represents the relational operators =, ≥, >, ≠ with the enumerated type {eq, ge, gt, ne} respectively. Conditions involving the other relational operators should be rearranged to use one of them;
2) constraint - represents the constraint of the branch predicate. The predicate should be reformulated to be in the form of c == 0 or c ≠ 0 or c ≥ 0 or c > 0;
3) *branch predicate* - The branch predicate in its original form;
4) *branch identification* - A number that exclusively identify the branch.

The `store` function records the values described above in global variables. These variables are indexed by the branch identification number. The identification number is also used as a stop condition to the numerical method. If a branch marked as the end of the path segment is reached, the iteration ends.

In the methodology proposed in this work, `store` is also used to control the program flow by selecting the way the branch predicate is executed. The branch identification numbers are given in the instrumentation *false* rather than during the program execution as done in [16], facilitating the instrumentation of loop conditions. The `store` function is presented in figure 4.

```c
bool store(comp_type type_v,
  int g_v, bool p_v, int id)
{
  type[id] = type_v;
  g[id] = g_v;
  p[id] = p_v;
  if (p_v) pTrue[id] = true;
  else pFalse[id] = false;
  if (id == stopBranch)
    segmentEnd();
  return (p[id] && !contExec) ||
      (contExec && Branch[id]);
}
```

Fig. 4. Store function

Variables `pTrue` and `pFalse` record if the branch has been executed in its true and false conditions respectively. They are used to determine what DFUs were not covered properly during the Functional Testing.

A signal controls if the branch is executed normally or forced. If the global variable `contExec` is *false*, the branch predicate is tested normally and the program flow depends on the input vectors. However, if `contExec` is *true*, the program flow is controlled by the values on the Branch boolean array. Each element of this array indicates if a branch should be forced to execute. The `contExec` is *true* while the optimization method is executing, allowing the program to be *execute* as a straight line through a path determined by the `Branch` array values.

V. NUMERICAL METHOD FOR TEST DATA GENERATION

Instead of symbolically executing the DUV, a numerical optimization method is used to determine the necessary input values to exercise a given path.

All decision statements in the DUV source code are substituted by path constraints of the form \( c_i = 0, c_i \neq 0, c_i \geq 0 \) or \( c_i > 0 \), where \( c_i \) is a real-valued constraint that measures how close the \( i \)th path condition is to being satisfied. If a statement predicate is not in one of the forms gived above, it should be rearranged. For example, a predicate of the form \( \text{ctl} < 2 \) should be rewritten as \( 2 - \text{ctl} > 0 \).

Using this path constraints, the predicate can be solved without explicitly knowing its exact form by *picking* values for the inputs such that \( c_i \) satisfies the given constraint. The constrained optimization problem can be formulated as minimizing the following function:

\[
f(x), x \in \mathbb{R}^n
\]

Subject to the following set of nonlinear path constraints:

\[
g_i(x) \begin{cases}
  > & \text{if } i \neq 0 \\
  \geq & \text{if } i = 0 \\
  \neq & \text{if } i \neq 0
\end{cases} 0, \quad i = 1, 2, ..., m
\]

Where the vector \( x \) represents the DUV \( n \) inputs and \( f(x) \) is a function which result represents the distance from the current vector \( x \) to the optimal vector.

Using a penalty function technique, the constrained optimization problem can be reduced to an unconstrained one with the following objective function:

\[
f'(x, w) = \sum_{i} G(g_i(x), w_i, type_i)
\]

Where \( w \) is a set of positive weighting factors and the term \( G(g_i(x), w_i, type_i) \) represents the penalty imposed upon the \( i \)th path constraint with a constraint value of \( g_i(x) \) and a relational operator represented by \( type_i \). The penalty function \( G \) is always positive and it is chosen such that its value is small if the constraint is satisfied. The value of \( g_i(x) \) is calculated forcing the execution of the branches in the \( i \)th path. Path constraints involving logical operators can be reduced using the following rules [16]:

- \( G(p_1 \text{AND} p_2) = G(g_1) + G(g_2) \)
- \( G(p_1 \text{OR} p_2) = \min(G(g_1), G(g_2)) \)
- \( G(\text{NOT} p) = G'(g) \)
- \( G(p_1 \text{XOR} p_2) = \min(G(g_1) + G'(g_2), G'(g_1) + G(g_2)) \)

Where \( p_i \) is a predicate with contraint value \( g_i \) and \( G'(g_i) \) is the penalty function to test the inverse relational operator in \( p_i \). The value of the vector \( x \) that minimizes function 3 can be calculated using any unconstrained multivariate optimization method such as a quasi-Newton using BFGS update. A more complete description of the Numerical Method for Test Data Generation employed here can be found on [16].

VI. A SIMPLE EXAMPLE

The piece of code in figure 5 is a simple arithmetic unit designed to calculate an operation over two integers. Possible operations are sum, subtraction, multiplication and division. There are three input variables for the design: \( a, b \) and \( \text{ctl} \), respectively the first operator, second operator and the operation selection line. Multiplication and division were designed as loop structures to illustrate the use of them in the
methodology. A specialization was made on multiplication and division by two. If the second operator is two, the operation is treated as a bitwise shift.

Analysing carefully the code, one can find two mistakes:

1) The specialization for the division on line 27 is mistaken. Division by two is the same as a shift right operation;

2) Again in the division operation, when the operators have different signals, the result would be wrong. There is a fault of omission in the code, as the signals of operators should be tested in order to determine the signal of the result.

```
1: if (store(eq, ctl - 1, ctl == 1, 1))
2: { c = a + b; }
4: )else{
5: if (store(eq, ctl - 2, ctl == 2, 2))
6: { c = a - b; }
8: )else{
9: if (store(eq, ctl - 3, ctl == 3, 3))
10: { if (store(eq, b - 2, b == 2, 4))
12: { c = a <= 1;
14: )else{
15: c = 0;
16: i = 0;
17: for (i = 0; store(gt, b - i, i < b, 5); ++i)
18: { c += a;
20: }
21: }
22: )else{
23: if (store(eq, ctl - 4, ctl == 4, 6))
24: { if (store(eq, b - 2, b == 2, 7))
26: { c = a <= 1;
28: )else{
29: if (store(ne, b, b != 0, 8))
30: { c = 0;
32: int a_v = abs(a);
33: int b_v = abs(b);
34: while (store(gt, a_v, a_v > 0, 9))
35: { a_v -= b_v;
37: c += c;
38: }
39: )else{
40: error = DIVISION_BY_ZERO;
41: }
42: }
43: }
44: }
45: }
46: }
```

For the Black-box testing, the input vectors are determined randomly, trying to cover all possible operations in the DUV. Using these vectors, detecting the fault of omission in the division code is easy, but to find the error of the division by two is not so simple as only when \( b = 2 \) and \( ctl = 4 \) the mistaken code is exercised.

The TFG for this example is represented in figure 6. The numbers in the vertices represent the branch identification numbers passed to the store function. The path not exercised by the Black-box testing vectors listed on table I is highlighted indicating the path to the uncovered DFU (branch 7 true). If the Verification engineers don’t know the specialization for the division by two, the line 27 of the code would not be verified by these vectors and the fault would not be detected.

The path constraints can be determined walking the path from the beginning to its end. However, the extraction of these constraints is not necessary to the methodology, as the path would be forced to executed as a straight line by the instrumentation. Each time the store function is called, it records the value of the constraint. The numerical optimization method is executed between every running of the path, using the constraints recorded values to determine the next input vector. The path constraints are listed below and the results for the firsts and lasts iterations are given on table II.

1) \( ctl - 1 \neq 0 \);
2) \( ctl - 2 \neq 0 \);
3) \( ctl - 3 \neq 0 \);
4) \( ctl - 4 = 0 \);
5) \( b - 2 = 0 \).

Analysing the results, one can verify that the input \( a \) does not influenciate the constraints of the path and that the input

<table>
<thead>
<tr>
<th>Input a</th>
<th>Input b</th>
<th>Input ctl</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>-5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-12</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>-6</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>-2</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**TABLE I**
BLACK-BOX TESTING VECTORS

Fig. 6. Coverage Flow Graph of the Example

Analysing the results, one can verify that the input \( a \) does not influenciate the constraints of the path and that the input
values to exercise the path are $b = 2$ and $ctl = 4$, as expected.

VII. CONCLUSION

A working methodology to generate test vectors for SystemC designs was presented. The methodology is complementary to the Functional testing as it uses code coverage information to generate test vectors to cover the portions of code not exercised by the Black-box Testing. Instead of using symbolic execution, vectors are generated using a numerical optimization method. This approach does not suffer from restrictions related to symbolic execution such as defining array reference values and loop boundaries, as the code is really executed together with the optimization.

A simple example was given, demonstrating the use of the methodology. Although the achieved results are stimulating, more studies are needed to determine the restrictions to the methodology and the impact of its use in the SoC design cycle.

Future work will concentrate on a tool to automatically instrument the SystemC source code and build the TFG. Methodologies to choose the best path to a vertex given the TFG and the path’s constraints will also be a matter of researched.

REFERENCES


