Rule-based Equivalence Checking of System-level Design Descriptions

Hiroaki Yoshida[1,2] Masahiro Fujita[1,2]

1. VLSI Design and Education Center (VDEC), University of Tokyo
2. CREST, Japan Science and Technology Agency

Abstract— **This paper presents our study on rule-based equivalence checking of system-level design descriptions. The rule-based equivalence checking proves the equivalence of two system-level design descriptions by applying equivalence rules in a bottom-up manner. In this paper, we first introduce our intermediate representation of system-level design, and then show a set of representative equivalence rules. Since our equivalence checking method is based on potential internal equivalences identified by using random simulation, we also present how to prove the equivalence based on such potential internal equivalences. Finally, we explain our implementation of the rule-based equivalence checker and demonstrate its feasibility and efficiency using an example design.**

I. Introduction

The recent dramatic advance of the semiconductor technologies leads to a large-scale system on a VLSI chip. As a consequence, the design cost and turn-around time have been increasing rapidly. Recently system-level design, which has a higher level of abstraction than register-transfer-level design, attracts more attention as an effective design approach. In the system-level design methodology, an entire system is described in a system-level design description language. Since it enables a hardware and software co-design, a design reuse, and a less description effort due to a higher abstraction, a higher design productivity can be achieved. As system-level design description languages, C-based languages such as SpecC [1] and SystemC [2] are widely used. CAD technologies are the key to a successful system-level design, and hence the subjects are being studied actively.

Apart from these advantages of the system-level design, if a design error is found in a later phase of the design flow, a designer needs to fix the design at the system level. This leads to the increase of the design cost and turn-around time. Thus, it is critical to find any design errors in an early phase of the design flow. To this end, formal verification is preferable to simulation-based verification which may fail to find corner-case errors. Although several formal verification techniques for system-level design descriptions have been studied in a number of literatures [3], [4], more efficient methods are always desired.

In the current system-level design methodologies [5], [6], they provide the tools which interactively support the transformations and refinement of a design. In these methodologies, a design is locally and incrementally refined by transforming the design manually. Thus, design errors can be efficiently found in an early phase of the design flow by checking the equivalences for every local transformation.

For this purpose, an rule-based equivalence checking method for system-level design description has been proposed [7], [8]. The rule-based equivalence checking of system-level design descriptions proves the equivalence of two design descriptions by applying a set of local equivalence rules in a bottom-up manner. By regarding each equivalence rule as a pattern in a graph, the rule-based equivalence checking can be regarded as a graph matching problem. Using the extended system dependence graph ExSDG [9] as an internal representation, an efficient implementation of the equivalence checking method is achieved.

Since the rule-based equivalence checking method proves the equivalences in a bottom-up manner, the equivalences of the variables need to be determined in advance. The equivalences of the variables, called as internal equivalences, are used to prove the equivalences of the expressions and statements using the variables. One possible solution to this problem is to identify the internal equivalences according to their names. However, the internal equivalences based on the names fail to prove the equivalences in the following cases:

- Variables are swapped
- Variable names are changed
- Intermediate variables are introduced

The examples in Fig. 1 illustrate these cases. Although the designs in Fig. 1 (a)-(d) are all functionally equivalent, the equivalence checker based on the names cannot prove the equivalence. Since the design modifications illustrated above are typical, it is critically important to prove the equivalence even in the presence of those modifications.

In this paper, we first introduce our intermediate representation of system-level design, and then show a set of representative equivalence rules. Then, we present our equivalence checking method which identifies potential internal equivalence and proves the equivalence based on such potential internal equivalences. Finally, we explain our implementation of the rule-based equivalence checker and demonstrate its feasibility and efficiency using an example design.

II. ExSDG: Extended System Dependence Graph

System Dependence Graph (SDG), proposed by Horwitz *et al.* [10], has been used mainly for program slicing. In an SDG, each node corresponds to a statement or an expression in a design description, and each edge represents a dependence relation between a pair of nodes. The dependence relation is categorized into three types: data dependence, control dependence, and declaration dependence.

```
int ex1(int a, int b) {
    return a - b;
}
```

(a) Original design

```
int ex2(int b, int a) {
    return b - a;
}
```

(b) Swapping variables

```
int ex3(int c, int d) {
    return c - d;
}
```

(c) No name correspondences

```
int ex4(int a, int b) {
    int c = a - b;
    return c;
}
```

(d) Introducing an intermediate variable

Fig. 1. Examples of rule-based equivalence checking based on the names.

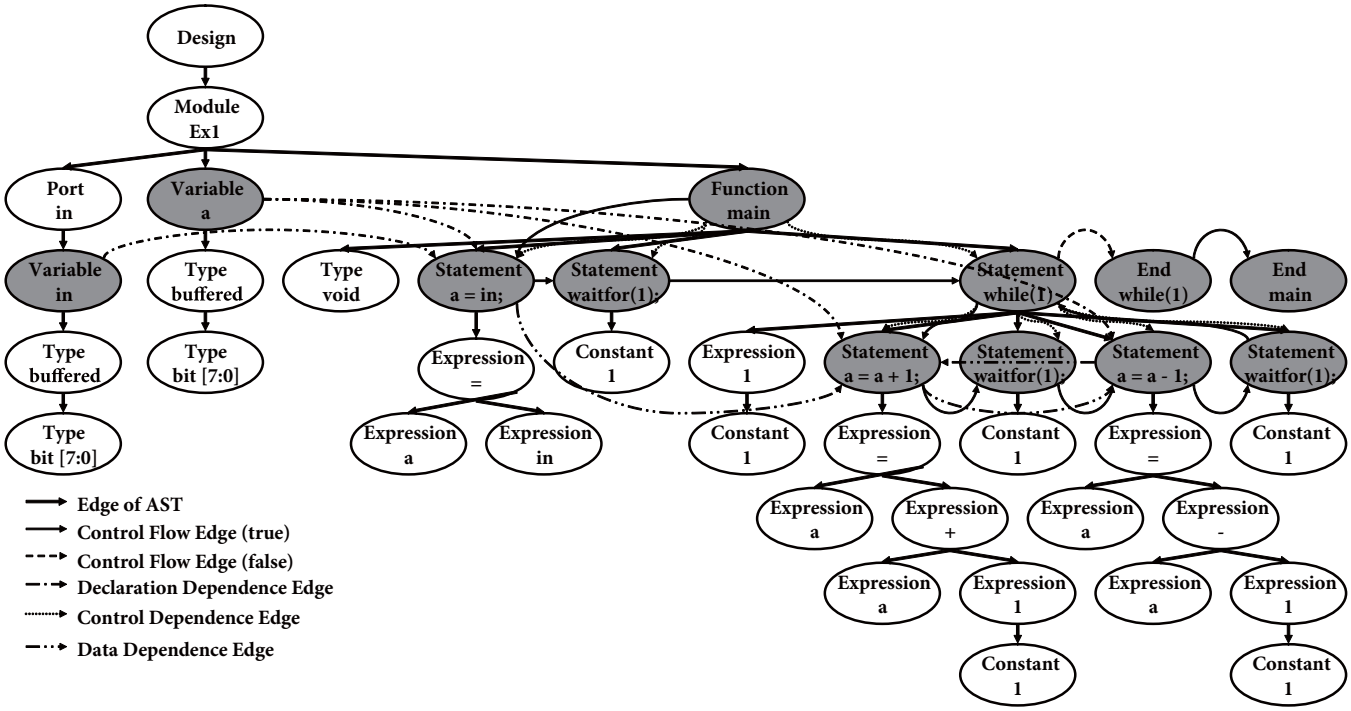

Fig. 2. ExSDG: Extended System Dependence Graph [9].

Data dependence There is a data dependence from a statement $s1$ to another statement $s2$ if a variable assigned in $s1$ is used in $s2$ without any other assignment to $s1$.

Control dependence There is a control dependence from a statement $s1$ to another statement $s2$ if $s1$ controls the execution of $s2$.

Declaration dependence There is a declaration dependence from a statement $s1$ to another statement $s2$ if a variable declared in $s1$ is used in $s2$.

Extended System Dependence Graph (ExSDG) is an abstract syntax tree with SDG dependence edges [9]. By representing a system-level design description using ExSDG, the manipulation and analysis of a system-level design can be performed efficiently. An example of an ExSDG is illustrated in Fig. 2.

III. Representative Equivalence Rules

Rule-based equivalence checking proves the equivalence of two designs by applying a set of pre-defined equivalence rules. Each equivalence rule is defined based on a static dependence relation and control flow. It is assumed that the equivalences of variables and constants in two design descriptions are already known. Given the ExSDG representations of two designs, each equivalence rule can be viewed as a pattern in an ExSDG. Therefore, the problem of the rule-based equivalence checking can be viewed as a matching problem of two ExSDGs. Note that a set of equivalence rules are chosen empirically and hence the proposed method is not guaranteed to prove any equivalences. We explain some important rules below. In the rules, $\equiv$ represents a design equivalence, $\triangleq$ represents a definition equivalence.

Rule 1 ***(Expression)*** *Two expressions are equivalent if the ASTs corresponding to the expressions are equivalent. The equivalence of ASTs include the arithmetic operation properties such as commutativity, associativity and distributivity (e.g. $c * (a + b) \equiv (b + a) * c$), and the properties of equality, inequality (e.g. $x < y \equiv y > x$).*

Rule 2 ***(Assignment)*** *Given an assignment statement N_{A2}, let N_V be a variable in the right-hand side of N_{A2} and N_E be an expression. Then, $N_V \equiv N_E$ if the following conditions are satisfied.*

1) *The right-hand side of N_{A1} and N_E are equivalent where N_{A1} be an assignment statement defining N_V.*
2) *No variables in the left-hand side of N_{A1} are redefined between N_{A1} and N_{A2}.*

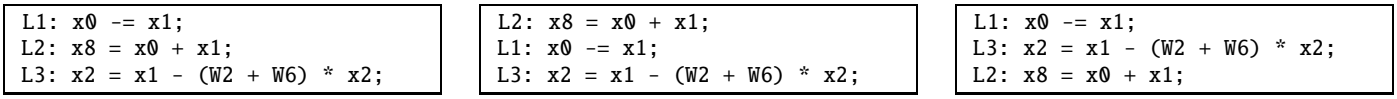

Fig. 3. Examples of equivalence checking using the sequential composition rule.

Rule 3 ***(Selection)*** *A conditional statement, like* `if` *statement in C language, consists of a condition expression, then block and else block. Let two conditional statements* S_1 *and* S_2 *be* $S_1 \triangleq$ ***if*** C_1 ***then*** N_{T1} ***else*** N_{E1} *and* $S_2 \triangleq$ ***if*** C_2 ***then*** N_{T2} ***else*** N_{E2}*, respectively. Then,*

Rule 3.1 $C_1 \equiv C_2 \wedge N_{T1} \equiv N_{T2} \wedge N_{E1} \equiv N_{E2} \Rightarrow S_1 \equiv S_2$

Rule 3.2 $C_1 \equiv \neg C_2 \wedge N_{T1} \equiv N_{E2} \wedge N_{E1} \equiv N_{T2} \Rightarrow S_1 \equiv S_2$

Rule 4 ***(Sequential composition)*** *In imperative programming languages such as C language, a series of statements are executed sequentially. The order of the statements can be changed without changing its functionality if there is no data dependency between them. Let two sequential compositions of statements be* $P_1 \triangleq N_{1,1}; ...; N_{1,n}$ *and* $P_2 \triangleq N_{2,1}; ...; N_{2,n}$*. Then,* $P_1 \equiv P_2$ *if the following conditions are met:*

1) *There exists a permutation map* π *such that* $N_{1,i} \equiv N_{2,\pi(i)}$ *for every i.*
2) *For every data dependence* $N_{1,i} \rightarrow N_{1,j}$ *in* P_1*,* $N_{2,\pi(j)}$ *is executed after* $N_{2,\pi(i)}$ *in* P_2*.*
3) *For every data dependence* $N_{2,\pi(i)} \rightarrow N_{2,\pi(j)}$ *in* P_2*,* $N_{1,j}$ *is executed after* $N_{1,i}$ *in* P_1*.*

For instance, in Fig. 3 (a), there exists a data dependence from L1 to L2 through `x0`*. Therefore, the program (b) obtained by swapping* `L1` *and* `L2` *is not equivalent to the program (a). In contrast, the program (c) obtained by swapping* `L2` *and* `L3` *is equivalent because there is no data dependence between them.*

Rule 5 ***(Loop)*** *For every loop statement such as* `for` *statement and* `while` *statement, it is assumed that such a loop are unrolled. Hence, the equivalence of two loops can be checked by the sequential composition rule.*

Rule 6 ***(Parallel composition)*** *In system-level design description languages such as SpecC, they provide a* `par` *statement for specifying a parallel composition of multiple statements.*

Rule 6.1 (Parallel-parallel) *Let two parallel compositions be* $C_{P1} \triangleq$ ***par***$\{N_{1,1}; ...; N_{1,k}\}$ *and* $C_{P2} \triangleq$ ***par***$\{N_{2,1}; ...; N_{2,k}\}$*. Then,* $C_{P1} \equiv C_{P2}$ *if there exists a permutation map* π *such that* $\forall i, 1 \leq i \leq k, N_{1,i} \equiv N_{2,\pi(i)}$*.*

Rule 6.2 (Sequential-parallel) *Let sequential and parallel compositions be* $C_S \triangleq N_{1,1}; ...; N_{1,k}$ *and* $C_P \triangleq$ ***par***$\{N_{2,1}; ...; N_{2,k}\}$*. Then,* $C_S \equiv C_P$ *if there exists a permutation map* π *such that* $\forall i, 1 \leq i \leq k, N_{1,i} \equiv N_{2,\pi(i)}$ *and there is no data dependence* $N_i \rightarrow N_j$ *for every i,j* $(i \neq j)$*.*

IV. Equivalence Checking Method

A. Identifying Potential Internal Equivalences Using Random Simulation

As explained in the previous section, the rule-based equivalence checking applies the equivalence rules in a bottom-up manner under the assumption that the equivalences of the variables between two designs, called as internal equivalences, are known. One possible solution is to identify the internal equivalences based on the names of the variables, *i.e.*, the variables with the same name are identified to be equivalent. However, as explained in Fig. 1, identifying the internal equivalences based on the variable names often fails the verification of designs before and after typical design modifications.

Our method identifies potential internal equivalences using random simulation. After simulating two target designs with randomly-generated input patterns, the variables with the same value are identified as potential internal equivalences. Potential internal equivalence identification methods using random simulation have been commonly used in equivalence checking of gate-level circuits [11]. We formulate a problem of identifying potential internal equivalences of system-level design descriptions as follows:

Problem 1 *Find a set of equivalence classes of the nodes corresponding to the variables in the ExSDG representations of two designs.*

An equivalence class is defined as a set of nodes which are equivalent to a representative node r:

$$[r] \triangleq \{n \in N_1 \cup N_2 \mid r \equiv n\}$$

where N_1 and N_2 are the sets of the nodes in two designs, respectively.

Fig. 4 describes the basic procedure for identifying potential internal equivalences using random simulation. We explain each step in this procedure as follows. First, each ExSDG is converted into a static single assignment (SSA) representation. Since every variable in a function is guaranteed to be assigned only once, it enables the identification of more potential equivalences. Next, the primary input ports in the designs are identified and then the input pattern for each primary input port is generated randomly. In this paper, we define a *cycle* as a period of the execution from accepting an input pattern to generating a set of output values. It is assumed that given designs can be executed on a per-cycle basis. Alternatively, one can specify a cycle as N times of handshake communications or an execution in N unit-time and so on.

Inputs: ExSDGs of two designs
Output: A set of equivalence classes of ExSDG variables

1: Convert each ExSDG into SSA representation
2: Identify the primary input ports
3: Generate a pattern for each primary input port randomly
4: Insert a monitor immediate after each assignment statement
5: Compile and execute the simulator
6: Generate the signature for each variable
7: Compute a set of equivalence classes based on the signatures

Fig. 4. Basic procedure for identifying potential internal equivalences using random simulation

Next, a monitor statement is inserted immediate after each assignment statement. This dumps the value of a variable in the left-hand side of the assignment statement. If a function is called multiple times in a cycle, the same variable is assigned multiple times. This also occurs when a module is instantiated multiple times. Though one can perform "elaboration" to duplicate modules and/or functions to distinguish these assignments, it is not efficient. Therefore, different assignments to the same variable in a cycle are distinguished by dumping not only the variable name but also their runtime context information. The runtime context information represents a path from the top-level module to the current function. A variable with such a context information is called as an extended variable. Every extended variable is guaranteed to be assigned only once in a cycle. If an extended variable is not assigned in a cycle, the variable has an invalid value.

Then, a pattern generator module for the random input patterns is created and connected to each design. By compiling the design description, a simulator is generated and executed. Then, the signature for each variable is generated based on the simulated values of the variable. A signature is defined as a vector of the values of an extended variable where the size of the vector is the number of the simulation cycles. In a software implementation, a signature can be represented as a string concatenating the strings of the values with a delimiter character. The variables with the same signature are potentially equivalent, and those with the different signatures are not equivalent. Note that *potentially equivalent* means that the variables are equivalent for the input patterns. Two extended variables with multiple contexts are potentially equivalent if they have the same signature for every context. Finally, a set of equivalence classes is formed by collecting the variables with the same signature.

B. Proving Equivalences Based on Potential Internal Equivalences

Once the potential internal equivalences are identified in Section IV-A, the equivalence checking in Section III can be performed even for the examples in Fig. 1. Since the potential internal equivalences are based on the simulation, they may include false equivalences. In this section, we propose an extension of the rule-based equivalence checking method, which works even in the presence of false internal equivalences.

Inputs: ExSDGs of two designs G_1, G_2
Potential internal equivalences
Output: A set of equivalence classes of ExSDG nodes

1: **repeat**
2: **for all** node N_1 in G_1 **do**
3: Apply the rule corresponding to the type of N_1
4: **if** found an equivalence **then**
5: Add N_1 and $N_2 \in G_2$ in an equivalence class
6: **else if** found a contradiction in a potential equivalence **then**
7: Remove the false equivalence from the equivalence class
8: Remove any equivalences derived the false equivalence from the equivalence class
9: **end if**
10: **end for**
11: **until** any equivalence classes have been updated in the last iteration

Fig. 5. Basic procedure for equivalence checking based on potential internal equivalences.

It is possible that no equivalence is found for a variable because the generated input patterns do not satisfy a condition and hence a portion of the program is not executed during the simulation. This issue is avoided by using an existing technique which generates the input patterns activating a specific portion of a program [12]. Therefore, we assume that the potential equivalences of every variable are identified.

Thus, there are only two possible cases: (a) the internal equivalences are all true and (b) some variables in a equivalence class are not equivalent. Note that no two variables in different equivalence classes are equivalent. The proposed equivalence checking method resolves most of false internal equivalences and can prove the equivalence accurately. Fig. 5 presents the basic procedure for the rule-based equivalence checking based on potential internal equivalences. The inputs to the procedure are two ExSDGs G_1, G_2 corresponding to the target designs and a set of equivalence classes of the ExSDG nodes corresponding to the variables. For each node $N_1 \in G_1$, the rule corresponding to the type of N_1 is applied. If two nodes N_1 and $N_2 \in G_2$ are found to be equivalent by the rule, a new equivalence class is introduced for these two nodes. If a contradiction in an internal equivalence is found during applying the rule, the false internal equivalence is removed from the equivalence class and also all the equivalences derived from the false equivalence are removed. This process is repeated until any new equivalence is found in the last iteration. If the equivalences of all nodes in the ExSDGs is found, two designs are proved to be equivalent.

Fig. 6 illustrates an example of how a contradiction in a potential internal equivalence is found during the procedure. In this example, `b` and `c` in both (a) and (b) are identified to be equivalent unless the input patterns include `a`=123. In

```
int ex5(int a, int b) {
S1:    int c, d;
S2:    c = (a == 123) ? a : b;
S3:    d = b - c;
S4:    return d;
}
```

(a) Original design

```
int ex6(int a, int b) {
S1:    int c, d;
S2:    c = (a == 123) ? a : b;
S3':   d = c - b;
S4:    return d;
}
```

(b) Modified design
(`d = b - c` is modified to `d = c - b`)

Fig. 6. Example of how a contradiction in potential internal equivalences is found during the equivalence checking.

such a case, the equivalence class is identified as:

$$\{\texttt{ex5/b}, \texttt{ex5/c}, \texttt{ex6/b}, \texttt{ex6/c}\}. \quad (1)$$

Without considering a contradiction, the rule-based equivalence checker determines that two designs are equivalent. However, `ex5/S3` and `ex6/S3'` are equivalent only if there are two disjoint equivalence classes:

$$\{\texttt{ex5/b}, \texttt{ex6/c}\}, \{\texttt{ex5/c}, \texttt{ex6/b}\}. \quad (2)$$

In contrast, `ex5/S2` and `ex6/S2` are equivalent only if there are two disjoint equivalence classes:

$$\{\texttt{ex5/b}, \texttt{ex6/b}\}, \{\texttt{ex5/c}, \texttt{ex6/c}\}. \quad (3)$$

Since (2) and (3) contradict each other, these potential internal equivalences are removed and the procedure terminates the equivalence checking.

V. Implementation & Experimental Results

We have implemented the rule-based equivalence checking method by identifying potential internal equivalences in C++. It is implemented on top of a system-level formal verification framework *FLEC* [13]. FLEC generates an ExSDG by parsing a design description in SpecC language, generating an abstract syntax tree, and performing a dependency analysis. SpecC reference compiler [14] is used for generating a simulator.

Two SpecC descriptions of inverse discrete cosine transform (IDCT) before and after a set of practical design optimizations are used as example designs. The design before the optimization executes the processes for rows and columns sequentially. The design after the optimization executes them in parallel and also the variable names are changed during the parallelization. The rule-based equivalence checker based on the variable name could not determine the equivalence of two designs because the it could not find the equivalence of some variables due to the change of their names. Our method successfully proved that two designs are equivalent. The runtime was several seconds where most of the time was spent by the compilation of the simulator. From the fact that the runtime of the compilation of simulator is typically proportional to the size of the design and also the rule-based equivalence checking can be regarded as a graph matching problem, our method is applicable to large-scale problems.

VI. Conclusions

Rule-based equivalence checking of system-level design descriptions proves the equivalence of two system-level design descriptions by applying the equivalence rules in a bottom-up manner. In this paper, we present an equivalence checking method which identifies potential internal equivalence and proves the equivalence based on such potential internal equivalences. The experimental results on an example design demonstrated that the proposed method can prove the equivalence of the designs before and after a practical design optimization.

References

[1] R. Dömer, A. Gerstlauer, and D. Gajski. SpecC Language Reference Manual, Version 2.0. SpecC Technology Open Consortium, 2002.

[2] T. Grötker, S. Liao, G. Martin, and S. Swan. System Design with SystemC. Kluwer Academic Publishers, 2002.

[3] S. Abdi and D. Gajski, "A formalism for functionality preserving system level transformations," in *Proc. Asia and South Pacific Design Automation Conf.*, pp. 139–144, Jan. 2005.

[4] T. Matsumoto, H. Saito, M. Fujita, "Equivalence Checking of C Programs by Locally Performing Symbolic Simulation on Dependence Graphs," in *Proc. Int. Symp. on Quality Electronic Design*, pp. 370–375, Mar. 2006.

[5] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski, "System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design," *EURASIP Journal on Embedded Systems*, 2008.

[6] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara, "Specify-Explore-Refine (SER): From Specification To Implementation," in *Proc. ACM/IEEE Design Automation Conf.*, pp. 586–591, Jun. 2008.

[7] S. Shankar and M. Fujita, "Rule-Based Approaches for Equivalence Checking of SpecC Programs," in *Proc. ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, pp. 39–48, Jun. 2008.

[8] H. Yoshida and M. Fujita, "Improving the Accuracy of Rule-based Equivalence Checking of System-level Design Descriptions by Identifying Potential Internal Equivalences," in *Proc. IEEE Int. Symp. on Quality Electronic Design*, Mar. 2009.

[9] T. Nishihara, D. Ando, T. Matsumoto, and M. Fujita, "ExSDG : Unified Dependence Graph Representation of Hardware Design from System Level down to RTL for Formal Analysis and Verification," in *Proc. Int. Workshop of Logic and Synthesis*, pp. 83–90, May 2007.

[10] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 1, pp.26–60, Jan. 1990.

[11] F. Krohm, A. Kuehlmann, and Arjen Mets, "The use of random simulation in formal verification," in *Proc. Int. Conf. on Computer Design*, pp. 371–376, 1996.

[12] I. Bongartz, A.R. Conn, N. I. M. Gould and Ph. L. Toint, "CUTE: Constrained and Unconstrained Testing Environment," *ACM Trans. on Mathematical Software*, vol. 21, no. 1, pp. 123–160, 1995.

[13] Y. Kojima, T. Nishihara, T. Matsumoto, and M. Fujita, "FLEC: A Framework for System-level Debugging Support, Formal Verification and Static Analysis," in *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies*, pp. 341–346, Mar. 2009.

[14] SpecC Reference Compiler. `http://www.cecs.uci.edu/~specc/reference/`.