SCEMIT: A SystemC Error and Mutation Injection Tool

Peter Lisherness
University of California, Santa Barbara
California, USA
peter@ece.ucsb.edu

Kwang-Ting (Tim) Cheng
University of California, Santa Barbara
California, USA
timcheng@ece.ucsb.edu

ABSTRACT

As high-level models in C and SystemC are increasingly used for verification and even design (through high-level synthesis) of electronic systems, there is a growing need for compatible error injection tools to facilitate further development of coverage metrics and automated diagnosis. This paper introduces SCEMIT, a tool for the automated injection of errors into C/C++/SystemC models. A selection of 'mutation' style errors are supported, and injection is performed though a plugin interface in the GCC compiler, which minimizes the impact of SCEMIT on existing simulation flows. Experimental injected error detection results are presented for the set of OSCI SystemC Example Models as well as the CHStone C High-Level-Synthesis benchmark set. Aside from demonstrating compatibility with these models, the results show the value of high-level error injection as a coverage measure compared to conventional code coverage measures.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Verification*

General Terms

Design Verification

Keywords

high-level synthesis, SystemC, mutation, coverage

1. INTRODUCTION

High-level models are increasingly used in electronic verification and design, slowly displacing register-transfer-level (RTL) models in the same way that RTL models displaced gate-level models in the past. This shift is being driven in part by 1) more efficient simulation at higher levels of abstraction, 2) large amounts of legacy C code for many algorithms, 3) more flexible partitioning and easier integration in hardware/software co-design, and, most importantly, 4) availability of several commercial high-level synthesis (HLS) tools[1]. Despite this momentum, more experience and tools for verifying and validating high-level hardware designs are needed to ease wide-scale adoption.

The problems with verifying and validating a high-level design are twofold. First, conventional code coverage metrics applied to the high-level model do not translate directly into coverage of RTL, which is typically used as an intermediate representation in HLS flows. This can clash with established verification practices, for example when companies have RTL coverage sign-off requirements, but also begs the question of whether high-level code coverage metrics are adequate for measuring verification completeness. The second problem comes when a bug is encountered and must be traced back to its root cause. Debugging and diagnosis are fairly well understood for RTL, but new methods must be developed to map the diagnostic information to the high-level model.

In this paper we introduce SCEMIT, a C/C++/**S**ystem**C** **E**rror and **M**utation **I**njection **T**ool that was developed to facilitate further research and development on high-level coverage metric and diagnosis. While fault/error injection is commonly used as a model-based coverage metric, such as single-stuck-at in gate-level netlists or 'mutation' analysis[1] in high-level and RTL designs, it is also useful in the experimental validation of novel coverage and diagnosis techniques. A more thorough discussion on the use of SCEMIT for each of these purposes: model-based coverage, diagnosis validation, and coverage metric validation, is presented in Section 2 of this paper.

SCEMIT differs from existing mutation tools [2][3] and SystemC error injection tools [4][5] in that it uses a plugin interface to the GCC compiler to perform the error injection. As we explain in Section 3.1, this approach shares many advantages with the other approaches, and most importantly allows error injection with only minor modifications to the simulation environment. This, in conjunction with its reliance on freely available infrastructure such as GCC [6] and the OSCI SystemC [7] libraries, makes it ideally suited for deployment in research environments. SCEMIT also supports error injection across C, C++, and SystemC, whereas previous tools support only plain C [2] or limited injection in C++/SystemC [4][5].

DAC'10, June 13-18, Anaheim, California, USA

[1]Mutation analysis is a design-error injection technique originally introduced for software testing[2][8] that has recently been adopted for hardware verification in a commercial tool[3]. A more in-depth look at mutation analysis is given in Section 2.1.

The remainder of the paper is structured as follows: Section 2 describes how error injection can be used for coverage calculation and diagnosis. Section 3 details the use of SCEMIT in an existing build/simulation environment and provides an example usage flow. Experimental results from using SCEMIT on the OSCI SystemC examples and CHStone C HLS Benchmarks are presented in Section 4. Section 5 concludes and discusses future directions for this research.

2. HIGH-LEVEL ERROR INJECTION

Error injection plays an important role in verification, validation, and test, both as a coverage metric and as a way of evaluating other coverage metrics, diagnosis tools, and resiliency techniques. While low-level fault injection is preferable for modeling manufacturing defects and electrical bugs, design errors are best modeled at the abstraction level at which the design is specified. Section 2.1 describes the use of error injection as a verification and validation coverage metric in high-level designs, followed by an explanation of synthetic error injection for verification research purposes in Section 2.2.

2.1 Error-Model Coverage

While the use of fault injection as a coverage metric is the standard in manufacturing test, it has only recently come into use for verification. This change has been driven largely by the inadequacy of code coverage metrics borrowed from software verification, especially when applied to hardware designs. While it is far more time consuming than code coverage measures, high-level error injection adds a degree of granularity unmatched by other techniques.

Conventional code coverage metrics, such as statement coverage or condition coverage, are commonly used in verification of both software and RTL designs. While they are useful as a coarse, baseline coverage target, they exhibit a number of shortcomings: 1) errors of omission cannot lead to a lower coverage, 2) they do not require diversity of stimulus data[2], and 3) they do not reflect whether the information produced by executed statements propagates to the testbench checkers. Shortcomings 1 and 2 are partially addressed in hardware verification by the use of ad-hoc functional coverage bins, which require, for example, entering all states of an FSM or performing certain sequences of instructions in a microprocessor.

The third shortcoming of code coverage, error observability, is exacerbated in hardware designs due to their inherent concurrency. While software computations are typically performed on-demand for performance reasons, an 'always' block in a hardware model will execute whenever its sensitivity list is excited, regardless of whether the resulting computation is stored or discarded. Because of this, it is easier to achieve a high code coverage in a hardware design without performing meaningful testing. A number of dataflow[9] and observability coverage[10] metrics have been proposed to alleviate this by requiring that the result of computations propagate to a checker or observation point before considering them covered. Such model-free approaches are inherently approximate, do not have much higher granularity than non-propagation metrics, and also do not require stimulus data diversity (shortcoming 2 above). The problem of observability is even more pronounced when performing post-silicon validation, where checkers are far fewer in number due to the need to implement them in hardware or at the chip outputs.

These problems with code coverage are known in the software testing community. Error injection, or mutation analysis[8], was proposed, and has been thoroughly studied, as a finer-grained coverage measure. In mutation analysis, the original source code is *mutated* by one of a number of *mutation operators*, which inject errors such as changing an addition operator to a subtraction operator or replacing a variable in the LHS of an assignment with a constant. This *mutation* produces a *mutant* variant of the original source code, which is then run through the verification test suite. If the test suite cannot distinguish the *mutant* as erroneous, there are two possible meanings: 1) the test set is inadequate, as it cannot distinguish an intentionally buggy version of the source code from the original, or 2) the injected error produced a functionally equivalent piece of software. In the first case, the verification engineer is expected to improve the test set, adding a test to distinguish this *mutant* from the original. This is analogous to the stuck-at fault model at the gate-level, where an undetected fault either guides additional test generation or produces equivalent logic (such that no test can be generated). Section 3.3 provides an example of how SCEMIT can be used to compute mutation coverage.

2.2 Synthetic Error Injection

Lacking a design with real errors, as is often the case in verification research, it is necessary to inject design errors to evaluate the capabilities of a proposed verification approach. In particular, automatic diagnosis and diagnostic test pattern generation tools require designs with errors for experimental evaluation of their effectiveness. Online checking and resilience schemes are also commonly validated through error or fault injection experiments, but the faults are injected at a lower level of abstraction to better model electrical bugs. While low-level models such as structural netlists have well-defined fault models that can be used to evaluate diagnosis tools, no such error models exist for high-level designs. Instead, errors are often injected in an ad-hoc fashion (typically by the same person who developed the diagnosis tool), a method prone to bias even when performed with the best of intentions. Automated synthetic error injection, as performed by SCEMIT, can be used to combat the inherent bias in manual error injection by acting as an impartial selector of both error location and type.

Synthetic error injection can also be used to test coverage metric quality: A coverage metric can be seen as indifferent to a given error if, after removing from the test set of all tests that detect the error, the coverage score is unaffected. This indicates that the coverage metric in question did not perceive the value of those tests (hence the unaffected score), even though they were useful in detecting the injected error. This sort of analysis is used in the experimental results of Section 4 to highlight the shortcomings of code coverage in dealing with subtle design errors.

[2]By stimulus data diversity, we are referring to the values of variables and result of an expression. Code coverage metrics will tell us how many times a statement was executed, but not if the input values differed between those executions. Diversity of input values is an important aspect of testing reflected by neither code nor dataflow coverage metrics.

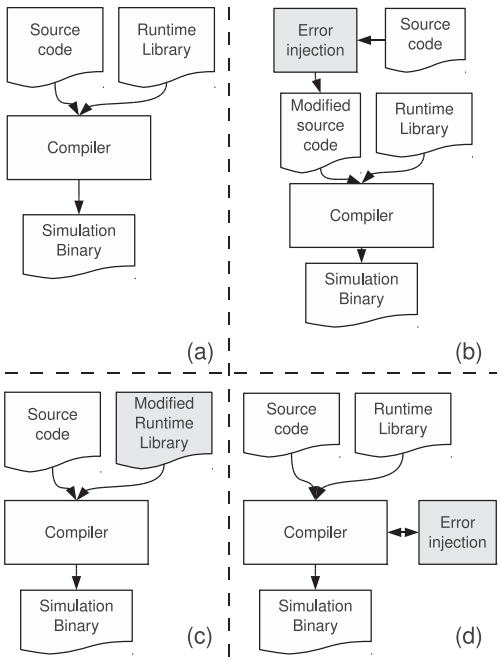

Figure 1: Simulator compilation flow and error injection schemes. a)original flow b)source modification c)modified runtime d)in-compiler injection

3. SCEMIT APPROACH AND DETAILS

While scalability is a primary concern in industrial tool use, where designs are large and usage cost is dominated by run time, academic research typically uses smaller designs and fewer tests, largely due to limited availability of suitable industrial-scale benchmarks. Experiment set-up time often dominates, so ease of integration and flexibility should be of primary concerns for new tool development. SCEMIT was built primarily to address these concerns, and therefore omits some of the advanced features, such as a GUI frontend or task-specific data collection, provided by other mutation analysis tools [3] [2].

SCEMIT is implemented as a plugin for GCC[6], and thus supports error injection in C, C++, and SystemC (using the OSCI[7] libraries) models. Injection is also possible in other languages with GCC frontends (VHDL, Ada, Java, Fortran, Lisp, etc.) but has not been thoroughly tested and may produce unexpected results. GCC version 4.5.0 or greater is required to use SCEMIT, as earlier versions do not include the plugin interface[3].

3.1 Error Injection Schemes

As previously stated, ease of integration was a motivating factor for SCEMIT. SCEMIT uses in-compiler error injection, as opposed to the code modification approach used by other mutation analysis tools [3][2] or the modified runtime solution used by other SystemC error injection tools [5][4]. These approaches and their effects on the existing simulation flows are shown in Figure 1.

Manual error injection typically follows the code modification style shown in Figure 1b, where the error is introduced into the original source before being passed on to the compiler. This approach benefits from flexibility because any compiler/simulator can be used. The primary drawback is that the injection tool must include a robust parser for source-to-source translation. This can be problematic when dealing with unportable/non-standards-compliant code and the idiosynchrosies of different target compilers: the parser needs the ability to interpret the source according to varied rule sets to gain the aforementioned flexibility. Source-to-source approaches also tend to impose many requirements on the build/simulation environment in order to allow substitution of the modified source for the original.

The modified runtime injection style, shown in Figure 1c, utilizes a modified simulation environment (e.g. modified libraries or instrumented code) to enable injection. The primary benefit of this approach is that it often requires only one compilation to add hooks for all errors, although this benefit is minimal when simulation time is significantly longer than compilation time. The drawbacks of this approach are: 1) it is limited to the specific compiler/simulator for which it was implemented, 2) the modified tools/libraries must be regularly updated as newer versions of the originals are released, 3) changes to the simulation flow are needed to force it to use the modified runtime, and 4) the error injection sites are limited to constructs instantiated or exposed from the runtime.

Our use of in-compiler injection, as shown in Figure 1d, represents a compromise between the other two approaches. The disadvantages are that it shares the code modification style's need to recompile for every error, and the modified runtime's limitation to a specific compiler/simulator. The other drawbacks of the previous approaches are overcome by integrating with the GCC compiler. The compiler preprocesses and parses the source code, eliminating the need for source-to-source translation, but the types of errors and available injection sites are similar to code modification. The use of the compiler's plugin interface allows the same environment to be used for both regular simulation and error injection, and reduces the need to update the injection tool as new versions of the compiler and libraries are released.

3.2 Available Error Models

The error models supported by this initial version of the SCEMIT tool are a simplified subset of the 'sufficient' mutation operators from [11], themselves a subset of those proposed in [8]. The most significant difference is the removal of many mutation operators that produce a great number of mutants, such as SSR (scalar-for-scalar replacement), which replaces LHS variables with all other variables of compatible type. Also, because SCEMIT operates on the intermediate representation of the code in the compiler rather than the source code itself, language- and syntax-specific mutation operators, such as those designed for ANSI C[12], are not meaningful. Note also that our set of errors does not include errors on the logical operators (&&, ||, i, as they are not normally present in the GCC internal representation. Instead, compound statements such as *if (a && b)* are decomposed into *if (a) if (b)* due to lazy evaluation. In light of all of these differences, some of the abbreviations and names used to identify the mutation/error models differ from those

[3]As of the time of this writing, version 4.5.0 of GCC is prerelease, available as source code from [6].

Table 1: Error/Mutant Types in SCEMIT

Name	Full Name	Example
OPR	Operator Replacement	a=b+1 ⇒ a=b-1
VCR	Var⇒Constant Repl.	if (a) ⇒ if (true)
CCR	Constant Replacement	a=b+1 ⇒ a=b+0
ROR	Relational Op. Replacement	a>=b ⇒ a==b

used in previous mutation research.

The mutation operators/error models supported by SCEMIT are summarized in Table 1, and described in greater detail in the following bullets:

- The OPR error type will replace the binary (i.e. taking two operands, as opposed to *unary*) operator in an expression with every other valid operator. The set of operators available in the GCC intermediate representation includes basic arithmetic, bitwise logic, and the min&max functions.

- The VCR and CCR error types replace an operand or value in the RHS of an assignment or the condition of a branch with a constant value. The constant currently takes *true* or *false* for branch conditions and values 0, 1, and -1 (or MAXINT for unsigned) for integer data types, but additional values can be forced if needed[4]. CCR additionally perturbs the existing constant value by +1 and -1 (again, additional values can be forced), and omits any replacements to the same constant value (i.e. it will not attempt to replace 0 with 0).

- The ROR error type replaces a relational or equality operator (==, !=, >=, <=, <, >) with every other equality operator, typically within the condition of a branch.

3.3 Mutation Coverage Flow Example

Figure 2 shows an exemplary flow for using SCEMIT to compute selective mutation coverage, limited of course to the subset of mutant operators supported by SCEMIT. Given an existing design built with the 'make' command and the test set executed with 'make check' that generates a golden output, the following steps would be needed:

1. The program is built and run *without* error injection enabled but *with* the SCEMIT plugin loaded using the appropriate compiler flags (typically with the CFLAGS or CXXFLAGS environment variables). The list of error sites is gathered from the build (output by default on STDERR) and a golden reference output is saved from the tests.

2. A shell script repeatedly invokes the build process and tests again, passing the indexes of each of the desired mutants to SCEMIT (again through compiler flags) and storing the resulting output. An execution timeout is generally needed in this loop, as some injected errors will prevent the simulation from halting.

[4]These values were selected as they are the identity values for many operations, and therefore have the effect of removing the replaced operand and operator entirely. Operator-deletion mutation is used by the commercial C++ mutation tool PlexTest[13].

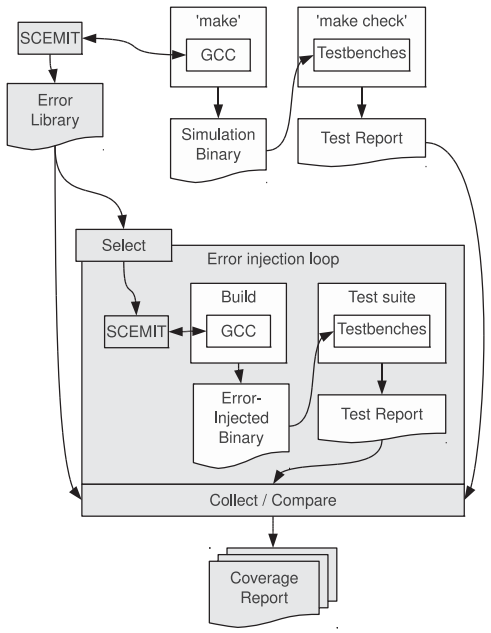

Figure 2: Flow for computing mutation coverage with SCEMIT. Items shown in gray are not part of the original simulation flow.

3. Once all errors have been injected and simulated, the 'diff' command is used to determine which of the errors caused an observable difference in the output.

While this is only a simple example, relatively few changes are needed to adapt it for different build/simulation flows or other types of analysis. Example scripts are provided with SCEMIT to serve as a template and allow users to recreate the experimental results presented in the following section.

4. EXPERIMENTAL RESULTS

In order to evaluate error injection with SCEMIT we selected a number of benchmark and sample designs, adapted them to the coverage measurement flow described in the previous section, and compiled the resulting error coverage results in Table 2. The first four designs are from the OSCI[7] SystemC 2.2.0 examples, while the remaining designs are the CHStone[14] C HLS benchmark set. The remaining columns in Table 2 summarize the total number of injected errors of each type, as well as the percentage detected by the testbench.

A few aspects of Table 2 are worth noting: First, the ROR errors typically exhibit lower detectability than the other error types. In many cases this is due to the injection of a functionally equivalent mutant, such as the replacement *for(x=0;x<10;x++) ⇒ for(x=0;x!=10;x++)*. This problem is well known in mutation testing, and much previous research has gone into detecting and excluding such equivalent mutants [15] [16]. Among the other three mutant types, detection levels of the included testbenches are on average much higher, but none particularly dominates. While high

Table 2: SCEMIT error injection results by error type

Error type:		All		OPR		VCR		CCR		ROR	
Design	Type	#	%Det.	#	%Det.	#	%Det.	#	%Det.	#	%Det.
fir	SysC	350	66	80	80	89	75	111	63	70	44
fir_rtl	SysC	468	85	152	100	192	91	69	70	55	38
fft_flpt	SysC	672	89	298	95	151	93	133	86	90	67
pipe	SysC	97	43	24	100	18	17	0	-	55	27
adpcm	C	3749	84	842	94	1983	84	744	78	180	63
aes	C	14742	29	4011	30	5014	29	5487	28	230	26
blowfish	C	11246	47	4432	46	3900	47	2824	47	90	20
dfadd	C	1779	60	432	56	750	69	307	52	290	50
dfdiv	C	1921	64	466	67	856	70	329	59	270	47
dfmul	C	1552	54	360	53	684	60	273	48	235	43
dfsin	C	3586	50	850	53	1601	54	565	53	570	32
gsm	C	4806	85	1160	88	2184	91	1172	77	290	60
jpeg	C	8069	30	2420	30	3304	31	1790	25	555	36
mips	C	1246	55	304	58	632	49	245	69	65	42
motion	C	1188	42	240	37	512	48	246	37	190	37
sha	C	1715	10	586	15	689	5	360	11	80	13

detectability isn't necessarily desirable, here it is more symptomatic of their lower proclivity to inject untestable errors.

Note also that the first two designs, *fir* and *fir_rtl*, implement the same functionality but differ in errors and error detection rate. More error injection locations were available in the RTL, which is expected as higher-level functional representations are typically more concise. However, even with a larger error population, the detection rates are higher. This suggests that the faults injected into RTL are easier to detect, but additonal experiments (and benchmarks at both levels of abstraction) are needed to determine the strength of this relation.

A subset of the CHStone benchmarks, listed in the first column of Table 3, were also modified to include additional test stimulus. This enables a closer look at the lack of granularity provided by the branch coverage. For each design, Table 3 lists the number of test vectors, the branch coverage, and the injected error coverage for both the original testbench and the additional random test stimulus. The remaining columns list the number of 'escapable' errors of each type. These are injected errors which are only detected by tests that *do not* provide a marginal increase in the branch coverage, so each one is a potential test escape if the tests are selected for branch coverage alone.

Note that while both forms of coverage increase with the additional stimulus, neither reaches (nor even approaches) 100%. Again, untestable errors are responsible for some of the undetected errors in this situation, and the presence of untestable branches (which were traced back to disabled round-off mode code in the designs) demonstrates the susceptibility of other coverage metrics to the problem of untestability. Also note that the escapable errors are greater than 10% of the total error populations in all cases. This is relatively unsurprising; despite their common use, control-flow coverage metrics are inadequate for many errors.

5. CONCLUSION

This paper introduces SCEMIT, an error injection tool designed to enable several applications in the verification of high-level hardware designs. By integrating directly with the GCC compiler, SCEMIT offers high compatibility with existing high-level simulation flows while still allowing fine-grained error injection. Our experimental results demonstrate compatibility with both SystemC and C models, and underscore the shortcomings of code coverage metrics in measuring verification completeness.

While SCEMIT is freely redistributable and open sourced under the GNU public license, it is not currently available on the web. Researchers interested in using SCEMIT should contact the authors of this paper for a copy, preferably via email with the name SCEMIT in the subject line. In the future, we hope to include additional error models into SCEMIT's repertoire, and encourage suggestions. Experiments using SCEMIT to determine relative quality of coverage metrics and error models across multiple levels of abstraction are currently underway, and we also intend to run additional experiments with more suitable SystemC designs as benchmarks become available.

6. REFERENCES

[1] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Des. Test*, 26(4):18–25, 2009.

[2] M. E. Delamaro and J. C. Maldonado. Proteum/im 2.0: An integrated mutation testing environment. pages 91–101, 2001.

[3] M. Hampton and S. Petithomme. Leveraging a commercial mutation analysis tool for research. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 203–209, Washington, DC, USA, 2007. IEEE Computer Society.

[4] N. Bombieri, F. Fummi, and G. Pravadelli. A mutation model for the systemc tlm 2.0 communication interfaces. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 396–401, New York, NY, USA, 2008. ACM.

[5] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto. Resp: a non-intrusive transaction-level

Table 3: Branch coverage sufficiency for injected errors

	Original Test			Orig. & Random Test			Escapable Errors				
Design	# Vecs	% Br.	% Err.	# Vecs	% Br.	% Err.	All	OPR	VCR	CCR	ROR
dfadd	46	64.7	52.7	1046	65.7	59.6	349	79	170	62	38
dfdiv	22	48.9	41.7	1022	68.2	64.4	290	97	140	31	22
dfmul	20	44.9	39.4	1020	68.0	53.6	190	51	95	28	16

reflective mpsoc simulation platform for design space exploration. In *ASP-DAC '08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, pages 673–678, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[6] Gcc, the gnu compiler collection. http://gcc.gnu.org/.

[7] Open systemc initiative (osci). http://www.systemc.org/.

[8] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, New York, NY, USA, 1980. ACM.

[9] T. Lv, J. Fan, and X. Li. An efficient observability evaluation algorithm based on factored use-def chains. In *Test Symposium, 2003. ATS 2003. 12th Asian*, pages 161–166, Nov. 2003.

[10] F. Fallah, S. Devadas, and K. Keutzer. Occom: efficient computation of observability-based code coverage metrics for functional verification. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 152–157, New York, NY, USA, 1998. ACM.

[11] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[12] H. Agarwal et.al. Design of mutant operators for the c programming language. Technical report, Purdue University, 1989.

[13] Plextest. http://www.itregister.com.au/products/plextest.htm.

[14] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 1192–1195, May 2008.

[15] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9:233–262, 1999.

[16] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Veri and Reliability*, 7:165–192, 1997.