Comparison of Model-Based Error Localization Algorithms for C Designs

Urmas Repinski, Jaan Raik

Department of Computer Engineering, Tallinn University of Technology, Raja 15, 12618 Tallinn, Estonia

ABSTRACT: The paper addresses *Model-Based Design Error Localization* in *C* designs. We consider a localization algorithm that is implemented with *Dynamic Slicing* and simulation using *C* code animation. The localization algorithm has been integrated into the FoREnSiC automated debugging system. Different ranking algorithms are compared and their ranking accuracy for Error Localization is measured by experimental results on the Siemens benchmark set. A new contribution of the paper is the observation that a simple error ranking metric that takes into account only information from failed sequences has the least average deviation from exact localization.

1 Introduction

In simulation-based verification, a digital system (design) is viewed as a *"black box"*, as it uses inputs and compares outputs, entire design structure is irrelevant. If we want to debug this design, then it should be transformed into a *"white box"* structure, or, in other words, it should be parsed into a representation suitable for processing. *"White box"* representation of the design is stored in corresponding data structure, in the model. A suitable model for *Error Localization* and *Correction* should allow tracing its execution and allow application of any changes within its structure. Localization algorithms use the trace of the executed model to select erroneous candidates from total components of model, correction require full access to any part of investigated design's model.

In this paper, implementation of the *Model-Based Error Localization* algorithm for *C* designs is presented. The algorithm has been integrated to the open source FoREnSiC [2] (FOrmal Repair ENvironment for SImple C) tool. The *FORENSIC* model has a flowchart-like structure that can be described as a special case of flowgraph: the *hammock graph* [4]. The tool includes a *Front-End* parser and a number of *Back-End Error Localization* and *Error Correction* algorithms' implementations.

Similar to [10], different ranking algorithms are compared and their ranking accuracy for error localization is measured by experimental results on the Siemens benchmark set. However, a new contribution of the paper is the observation that a simple error ranking metric that takes into account only information from failed sequences has the least average deviation from exact localization.

The paper is organized as follows. Section 2 explains the model-based error localization algorithm. Section 3 presents the different rankings used in error localization. In Section 4, the efficiency of these rankings is compared on the benchmark circuits. Finally, conclusions are drawn.

2 Model-Based Error Localization

2.1 Model Definition

In order to perform design *Verification* and further *Error Localization* and *Correction* of design, a model that is a suitable representation for *Error Localization* and *Correction* of the processed design that represents design as a *"white-box"* structure, should be developed. In the *FORENSIC* tool [2], the model of the design is a special case of *flowgraph* [4], known as *hammock graph* [4]. *Hammock graph* is defined as follows.

Definition 1: *Hammock graph* is a structure $H=<N, E, n_0, n_e>$, where N is a set of nodes, E is a set of edges in $N \times N$, n_0 is *initial node* and n_e is *end node*. If (n, m) is in E then n is an *immediate predecessor* of m and m is an *immediate successor* of n. A *path* from node n_0 to node n_e is list of nodes $p_0, p_1, ..., p_k$ such that $p_0 = n_0, p_1 = n_1, \ldots, p_k = n_e$, and for all i, $1 \le i \le k - 1$, (p_i, p_{i+1}) is in E. There is a *path* from n_0 to all other nodes in N. From all nodes of N, excluding n_e, there is a *path* to n_e.

In the *FORENSIC* tool, the C design is parsed into the flowgraph model using *gcc front-end* parser. Implemented model is designed with *Error Localization* and *Correction* in mind.

2.2 Simulation Implementation

Error Localization starts if the processed design fails its *Verification*, i.e. if output responses of the design do not match the reference responses in design's specification. *Simulation-Based Verification* requires simulation of the design. It is a widely-used approach to ensure functional correctness of hardware (and software) designs [5]. It is done by co-simulating a design under verification with an independently created specification and checking conformance of their reactions [6]. *Simulation* is responsible for getting output responses from input stimuli of the processed design model. Error Localization itself requires simulation of the model also,

978-1-4799-2096-9/13/$31.00 ©2013 IEEE

and simulation is the most important and time-consuming component of it.

In the *FORENSIC* tool [2], simulation is implemented using *C* functionality, by compiled-code execution of an instrumented code of the C design. Following is advantages of using C language's functionality for simulation of C design over simulation of the C design's model.

In the case of simulation of the C model, all logical operators should be implemented and all values of variables activated during simulation should be stored somewhere during processing. Variable's types can be very different: pointers, arrays, compound data types and so on, and memory model of C programming language should be rewritten in order to store the values. Decision to use already existing C compiler's memory model is therefore reasonable. Additionally, implementation of operators would be duplication of C functionality.

On the other hand, when using C functionality, then model should be dumped into a C executable file, instrumented, executed, and output responses should be written into output. This allows getting any data during simulation process, and we do not lose any functionality, however getting benefits such as speed of execution, simplicity of approach and we do not get any programming errors while complicated C functionality is re-implemented. Also *Dynamic Slicing* described in Section 2.4 is implemented in the *FORENSIC* tool [2] based on *C* functionality.

Of course the same principles of design instrumentation can be applied if the design is written using any other programming language, C/C++/SystemC or JAVA as examples.

2.3 Model-Based Error Localization

During simulation of the model during the *Model-Based Error Localization* number of nodes from model are activated.

Definition 2: *Activated path* P_s of *hammock graph* H=<N, E, n_0, n_e> is the path, that consists of nodes $n_j \in N$ simulated during simulation with input stimulus s. Path P_s from node n_0 to node n_e is a list $\{n_0, n_1, ..., n_{k-1}\}$, such that for all j, $0 \le j \le k-2$, (n_j, n_{j+1}) is in E.

Definition 3: *Activated nodes* $N_a \subset N$ during simulation with input stimulus s in the set of input stimuli S are nodes, that belong to the activated path P_s from node n_0 to node n_e.

Algorithm 1: Model-Based Error Localization. Model is simulated with input stimuli S and for each input stimulus $s \in S$ output responses of simulation are compared with reference responses of the specification. If the comparison, i.e. *Verification* fails, then corresponding *activated nodes* N_a have the *failed* counter increased, otherwise nodes have the *passed* counter increased. After simulation with all input stimuli S ranking algorithms are applied to counters, and finally nodes for all simulations with input stimuli S are sorted according to their *rank*. Nodes with high ranks are *Candidates for Correction* and stored in corresponding data structure.

Additional algorithm can be applied to increase *Error Localization* accuracy – *Dynamic Slicing* to *Activated Node*s for every simulation with input stimulus $s \in S$. In the next subsection, *Dynamic Slicing* will be described in details.

2.4 Dynamic Slicing for Error Localization

Dynamic Slicing is a technique that is applied to *Activated Nodes* N_a obtained during *Model-Based Error Localisation* simulation and allows reducing number of *Candidates for Correction*. The idea behind *Dynamic Slicing* is the following: some amount of candidates (nodes) are activated during *Model-Based Error Localisation's* simulation, but do not have any influence on simulation output. Those are usually constant declarations, assignments inside code and etc. *Dynamic Slicing* allows discarding those statements.

Definition 4: *Dynamic Slice* N_d of a *hammock graph* H=<N, E, n_0, n_e> includes nodes that are in activated nodes N_a that have influence on simulation output. Those nodes, that are *not in slice*, i.e. $N_a \setminus N_d$ can be removed from *Activated Path* of simulation with input stimulus s because they have no effect on simulation output.

In order to find nodes of model's *hammock graph* that are not in slice we need to introduce the following definitions — defined *DEF(n)* and referenced *REF(n)* variables in node n (class FlowChartNode in *FORENSIC* implementation [2]).

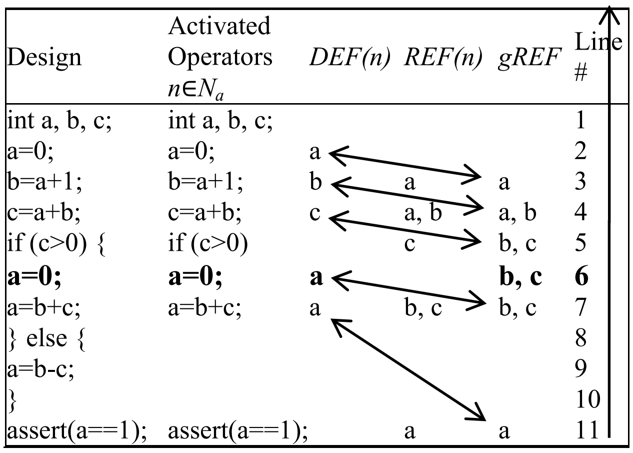
Definition 5: Let V be the set of variables, that exists in design's model represented by a *hammock graph* H=<N, E, n_0, n_e>. Then for every node $n \in N$, two sets can be defined, each of them is in V: *REF(n)* – set of variables, whose values are used in n, and *DEF(n)* – set of variables, whose values are changed or defined in n [3].

Algorithm 2: Dynamic Slicing. *Dynamic Slicing* algorithm uses *Activated Nodes* N_a obtained during simulation, global referenced variables *gREF* are calculated at the moment of processing of every component n that is in N_a. Execution of algorithm goes from end activated node n_e to initial activated node n_0, backtracing is performed. At the beginning of processing $gREF = \emptyset$.

If n contains a *C* function or condition (ConditionNode), then used variables *REF(n)* in n are added to the set of global referenced variables: $gREF = gREF \cup REF(n)$. *Output Observation Points*, that output data during simulation, can be last executed components during *Model-Based Error Localization* simulation and variables, that are outputted by *Output Observation Points* are initially stored in the set of global referenced variables *gREF*. Observation Points are implemented using *assert(...)* or special *FORENSIC_...()* functions.

If node *n* contains an assignment, then referenced variables *REF(n)* in the assignment are added onto the set of global referenced variables if and only if defined variables *DEF(n)* are already in the set of global referenced variables *gREF*, otherwise component is not in the slice and has no influence on the simulation result. Defined variables *DEF(n)* are removed from the list of global referenced variables, as they are redefined: if *DEF(n)* $\in$ *gREF* then *gREF* = (*gREF* \ *DEF(n)*) $\cup$ *REF(n)*, otherwise *gREF* = *gREF* \ *DEF(n)* and *n* is *not in the slice*.

Components that are *not in the slice* can be removed from *Activated Path* without changing the simulation output.



Design	Activated Operators $n \in N_a$	*DEF(n)*	*REF(n)*	*gREF*	Line #
int a, b, c;	int a, b, c;				1
a=0;	a=0;	a			2
b=a+1;	b=a+1;	b	a	a	3
c=a+b;	c=a+b;	c	a, b	a, b	4
if (c>0) {	if (c>0)		c	b, c	5
a=0;	**a=0;**	**a**		**b, c**	**6**
a=b+c;	a=b+c;	a	b, c	b, c	7
} else {					8
a=b-c;					9
}					10
assert(a==1);	assert(a==1);		a	a	11

Figure 1. Example of Dynamic Slicing execution, Assignment at Line 6 is not in slice.

Depending of processed design's nature, *Dynamic Slicing* can have high influence. Number of assignments can be made at the initializing part of design, those are usually constants, arrays, and so on, and not all of initialized values are used during simulation. Corresponding assignments will be discarded, as they will be not in slice. In actual implementation of *Dynamic Slicing* in *FORENSIC* tool [2], unique *C* variable addresses are used instead of variable names, that are shown at *Figure 1*, that were possible to implement only because of *C* functionality were used for simulation of model.

3 Ranking for Error Localization

Rankings are applied to model's *Activated Nodes* during *Model-Based Error Localization*; this allows extracting *Candidates for Correction*. Ranking equations and comparison of rankings using Siemens Designs [9] were studied in [10]. However, a new contribution of the paper is the observation that a simple error ranking metric that takes into account only information from failed sequences has the least average deviation from exact localization. The goal of this work was not to correct errors as in [11], but to compare coefficients for ranking algorithms.

Coefficients studied in this paper are taken from diagnosis/automated debugging tools Pinpoint[1], Tarantula[11], and AMPLE (Analyzing Method Patterns to Locate Errors)[8], and from molecular biology domain (Ochiai coefficient)[10].

A program spectrum [12] is collected in [10] of processed designs. Based on obtained spectra, that is *activated nodes* and *passed* and *failed* information in terminology introduced above, binary matrix *M x N* is built: *M* simulations of design of *N* blocks length[10]. Spectra can be presented as:

$$M\ Spectra \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}$$

Every component of spectra x_{ij} has value 1 if it were activated and 0 if it were not activated during simulation. Values e_j have error detection values information, have value 1 if simulation were *failed* and 0 if corresponding simulation were *passed*. Using notation of [7], four counters can be introduced:

	values:	
counter:	x_{ij}	e_j
a_{11}	1	1
a_{10}	1	0
a_{01}	0	1
a_{00}	0	0

(2.5)

Using counters similarity coefficients were calculated. Corresponding ranking algorithms using terminology introduced above were derived. Comparison of ranking definitions is in *Table 1* below.

Table 1. Comparison of ranking algorithms for FORENSIC Simulation-Based Back-End with experiments in [10].

[10] similarity coefficients s_j	FORENSIC tool ranking
Tarantula tool ranking[11]	
$s_j = \dfrac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}}$	$rank(n) = \dfrac{\frac{failed(n)}{totalfailed}}{\frac{passed(n)}{totalpassed} + \frac{failed(n)}{totalfailed}}$
Jaccard coefficient used in Pinpoint framework [10][1]	
$s_j = \dfrac{a_{11}}{a_{11}+a_{01}+a_{10}}$	$rank(n) = \dfrac{failed(n)}{passed(n) + totalfailed}$
Ochiai coefficient, used in molecular biology[10]	
$s_j = \dfrac{a_{11}}{\sqrt{(a_{11}+a_{01}) \cdot (a_{11}+a_{10})}}$	$rank(n) = \dfrac{failed(n)}{\sqrt{totalfailed \cdot (passed(n) + failed(n))}}$
AMPLE tool ranking[10]	
$s_j = \left\lvert \dfrac{a_{11}}{a_{11}+a_{01}} - \dfrac{a_{10}}{a_{10}+a_{00}} \right\rvert$	$rank(n) = \left\lvert \dfrac{failed(n)}{totalfailed} - \dfrac{passed(n)}{totalpassed} \right\rvert$
***simple* ranking introduced in current article**	
	$rank(n) = \dfrac{failed(n)}{totalfailed}$
Counters a_{ij} are defined in (2.5)	where *rank* of *Activated Node n* with simulation with all inputs *S* is calculated using *passed(n)* – times passed for node *n*, *failed(n)* – times failed for node *n*, *totalpassed* – total times passed for all inputs and *totalfailed* – total times failed for all inputs.

4 Experimental Results

Model-Based Error Localization accuracy experiments for various rankings were performed using *FORENSIC* tool [2]. The results are presented in *Table 3*, where the percentage of correction mutations required to apply in order to make correction with corresponding *Error Localization* ranking is used as the metric of accuracy. In article [10] similar experiments are described, but percentage of statements to be inspected to reach error location in design were used as meter of accuracy, results are in *Table 2*. Siemens Designs [9] were used as input designs in both experiments.

Table 2. Error localization accuracy using similarity coefficients in [10] – mean percentage of statements to be inspected to reach an error.

	Tarantula	Jaccard	Ochiai	AMPLE
print tokens	10.5	7.3	1.0	2.0
print tokens2	20.0	17.5	13.9	16.4
replace	12.2	11.6	7.6	12.7
schedule	3.0	2.9	1.6	11.3
schedule2	31.3	31.1	25.1	34.7
tcas	8.8	8.8	7.9	9.8
tot info	11.0	9.6	7.1	13.2

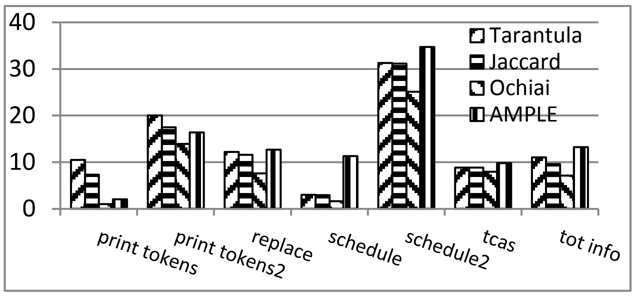
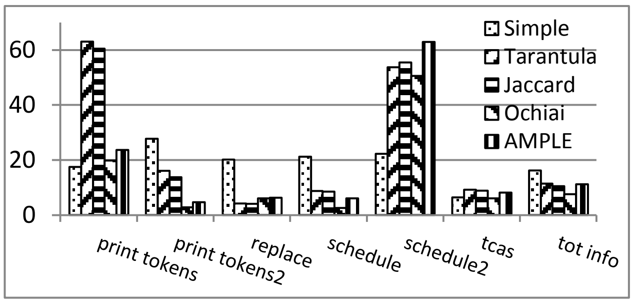


Table 3. Error localization accuracy using rankings in FORENSIC[2] – mean percentage of mutations required to be applied to correct an error.

	Simple	Tarantula	Jaccard	Ochiai	AMPLE
print tokens	17.46	62.97	60.42	19.84	23.59
print tokens2	27.73	16.08	13.69	2.84	4.65
replace	20.15	4.13	4.02	6.05	6.3
schedule	21.19	8.76	8.47	2.61	6.05
schedule2	22.21	53.65	55.41	50.51	62.86
tcas	6.35	9.17	8.89	6.01	8.16
tot info	16.23	11.36	10.42	7.53	11.21
Std. Deviation	6.61	24.04	24.03	17.27	20.99



Standard deviation of results, obtained with *FORENSIC*[14] tool is presented in *Table 3*.

5 Conclusions

In the article *Error Localization* algorithm is described that is implemented with *Dynamic Slicing* and simulation using *C* code simulation. The localization algorithm has been integrated into the FoREnSiC automated debugging system. Different ranking algorithms were compared and their ranking accuracy for Error Localization was measured by experiments on the Siemens benchmark set. A new contribution of the paper was the observation that a simple error ranking metric that takes into account only information from failed sequences has the least average deviation from exact localization.

Experiments presented in this work show that the *Simple* ranking for *Model-Based Error Localization* is more stable than other rankings introduced in [10], as it has lowest standard deviation of results – 6.61, that is 2.5 times less than second minor standard deviation of results, obtained using Ochiai ranking for *Error Localization*.

Simple ranking is best *when one error* is present in processed design, as in this case every failed simulation of design will include erroneous component, and it will have highest ranking. However, other rankings may become necessary if *multiple* errors are present simultaneously. Our future work is focused on studying error localization for multiple design errors.

Acknowledgements. The work was supported by FP7 project Diamond and Estonian Science Foundation.

References

[1] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: Problem determination in large, dynamic internet services, In Proc. Int. Conf. on Dependable Systems and Networks, pp. 595–604, 2002.
[2] FORENSIC tool, http://www.informatik.uni-bremen.de/agra/eng/forensic.php, 2012.
[3] Mark Weiser., Program Slicing, IEEE Transactions on Software Engineering, vol. SE-10, no. 4, pp. 352-357, 1984
[4] Kas'janov, V. N., Distinguishing Hammocks in a Directed Graph, Soviet Math. Doklady, vol. 16, no. 5, pp. 448-450, 1975.
[5] William K. Lam, Hardware Design Verification. Simulation and Formal Method- Based Approaches, Prentice Hall. Profession Technical Reference, 2005
[6] Kamkin, A, (2011) Simulation-based hardware verification with time-abstract models, Design & Test Symposium (EWDTS), 2011 9th East-West, pp. 43 – 47, 2011.
[7] A. K. Jain and R. C. Dubes, Algorithms for clustering data, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, pp. 528–550, 1988
[8] V. Dallmeier, C. Lindig, and A. Zeller, Lightweight defect localization for Java, In ECOOP 2005: 19th European Conference, Glasgow, UK, July 25–29, volume 3568 of LNCS, pp. 528–550, 2005.
[9] Bug Aristotle Analysis System -- Siemens Programs, HR Variants, http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/, 2012.
[10] Rui Abreu,, Peter Zoeteweij and Arjan J.C. van Gemund, An Evaluation of Similarity Coefficients for Software Fault Localization, Dependable Computing, PRDC '06. 12th Pacific Rim International Symposium on, pp. 39 – 46, 2006.
[11] Vidroha Debroy and W. Eric Wong, Using Mutation to Automatically Suggest Fixes for Faulty Programs, Int. conf. on Software Testing, Verification and Validation, (ICST), pp. 65 – 74, 2010.
[12] T. Reps, T. Ball, M. Das, and J. Larus, The use of program profiling for software maintenance with applications to the year 2000 problem, In Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), volume 1301 of LNCS, pp. 432 – 449, 1997.