Effective Model-Based Error Localization with Mutation-Based Error Correction in the C Designs

Urmas Repinski

Tallinn University of Technology

Department of Software Science

Tallinn, Estonia

urrimus@hotmail.com

Alex Norta

Tallinn University of Technology

Department of Software Science

Tallinn, Estonia

alex.norta.phd@ieee.org

Within contents of the FP7 DIAMOND project, effectiveness of error localization and error correction algorithms are investigated that are applied for the C designs. Already known ranking algorithms for the Model-Based Error Localization, e.g. Ample tool ranking, Jaccard ranking, Ochiai ranking and Tarantula tool ranking, are re-implemented using the developed FORENSIC tool. Additionally, we investigate the effectiveness of Consecutive and Simple ranking approaches together with the influence of the Dynamic Slicing algorithm for Model-Based Error Localization accuracy. The Mutation-Based Error Correction algorithm we use for error correction, while the number of mutations required to be applied for error correction we consider as a meter of the error localization and error correction accuracy. Experimental results, obtained with FORENSIC tool and compared with published in relative works, show the effectiveness of the suggested approaches and possibility for practical application in the software-development industry.

Keywords — Verification, specification, model, Dynamic Slicing, C design, Model-Based Error Localization, Mutation-Based Error Correсtion, simulation, animation.

# Introduction

Any software design (program) can be transformed to its equivalent hammock graph [2][3], with one start node that represents the beginning of the design, and one end node to denote an end of the design. Let us name this hammock graph structure – a "white box" representation of the design [11] – model of the design. This fact serves in the software-development industry for software testing when error-localization and error-correction algorithms are applied to the model. Error localization and -correction in software design are algorithms that are able to locate and correct errors in the design. FORENSIC tool [1][7] is a deliverable of the FP7 DIAMOND project and a tool for automatic error localization and correction in software designs.

In the tool termed Model-Based Error Localization algorithm [1], the re-implementation comprises a Simple and Consecutive rankings and already known Ample tool ranking, Jaccard ranking, Ochiai ranking and Tarantula tool ranking. Corresponding experimental data can be used for practical implementation in software development industry.

Additionally, in the FORENSIC tool [1][7] the Dynamic Slicing algorithm can render the Model-Based Error Localization algorithm’s implementation more accurate and faster, than the implementation without Dynamic Slicing. Corresponding experimental results show this.

Furthermore, the Mutation-Based Error Correction algorithm applies various correctional mutations to the most probable erroneous model's components in order to correct the error(s). Experimental results of the algorithm’s implementation in the FORENSIC tool [1][7] is compared with results in [10].

Various additional algorithms are applicable for model simulation, verification and specification animation for the error localization and error correction. Those algorithms are implemented in the FORENSIC tool and we refer the reader to [1][7] and this article for further information.

The goal of the FP7 DIAMOND project was to develop new algorithms and methods for error localization and error correction in the C designs, to implement them in a tool and to receive the experimental results, that can be used further for developing industrial error localization and correction tools. This research fills the gap and investigates the research question of how to implement error localization and error correction algorithms for practical implementation in software development industry? In order to answer this question the main research question is divided into three main questions: First, what is model for the C design? Second, what is Model-Based Error Localization algorithm? Third, what is Mutation-Based Error Correction algorithm?

The remainder of this article is structured as follows. Section 2 provides mathematical definitions and a graphical illustration of the model, used in the FORENSIC tool [1][7]. Section 3 describes the Model-Based Error Localization algorithm’s implementation in the FORENSIC tool [1][7] with mathematical definitions, algorithm’s description, model simulation approaches, rankings used in the algorithm and Dynamic Slicing’s algorithm’s implementation, followed by Section 4 that describes Mutation-Based Error Correction algorithm. Section 5 shows experimental results, obtained with the FORENSIC tool [1][7] that include Model-Based Error Localization, Dynamic Slicing and Mutation-Based Error Localization and comparison with relative data in [2][10][11]. Research is implemented within contents of the FP7-ICT-2009-4-248613 DIAMOND project.

# FORENSIC tool structure

Generally FORENSIC tool [1] is divided into two components – front-end and back-end, front-end parses design into the model, back-end provides Error Localization and Error Correctionfunctionalities.

## Front-End Implementation

Before starting process of Verification with Error Localizationwith Dynamic Slicing, based on the model, it is necessary to parse processed design into the model representation, that is task of front-end parser.

GIMPLE plug-in for GCC compiler [13] allows to obtain three-address representation of the processed design with tuples of no more than 3 operands. This representation of the design is suitable for model representation, as it has whole information about processed design. At this stage no need to create additional parser or lexical translator, C compiler’s plug-ins already generate suitable design form, and it is just necessary to store the parsed design tree into the model. For that every class of the model implementation has add() and remove() methods, using that new components to the model are added.

design

specification

inputs

simulation

outputs

model

parsing, front-end

equivalent?

Error Localization and Error Correction

reference outputs выводы

animation

NO

a)

b)

design

specification

inputs

parsing, front-end

animation

model

reference outputs

outputs

equivalent?

Model-Based Error Localization Dynamic Slicing, simulation

candidates for correction

Mutation-Based Error Correction

Corrected

model

simulation

corrected outputs

No valid

Correction found

equivalent?

Corrected Design, Location(s) of Errors

NO

NO

YES

Input/Output

Functionality

Structure

YES

Corrected Design, Location(s) of Errors

The front-end parser of the FORENSIC tool [1] for C designs were implemented in University of Bremen by Alexander Finder and colleagues from the Group of Computer Architecture. The front-end and entire tool can be extended to C++ and to SystemC support.

## Back-End Implementation

Various back-ends and algorithms are implemented in FORENSIC tool [1] for Error Localization and Correction, and Simulation-Based back-end's Error Localization with Dynamic Slicing, implemented in Tallinn University of Technology, is described in current article.

Simulation-based back-end's implementation consists of several components that are responsible for different functionality. Those are:

* Inputs-outputs, is data that is required and used for Simulation-Based Verification, Error Localization and Error Correction.
* Data Structures, are places where data information is stored during processing. This includes model, that is most significant data structure in tool.
* Functionality, is front-end, back-end, animation and simulation algorithms implementations, are implementations of Error Localization, Dynamic Slicing and Error Correction, that can be extended further.

## Tool Implementation

Diagram that describes basic Verification with Error Localization and Error Correction algorithm's implementation in FORENSIC tool [1] is Figure 1a and more accurate verification algorithm's implementation is Figure 1b.

Difference between two implementations is following: in a) implementation simulation of processed design is used to generate outputs, in b) implementation there is no independent design’s simulation with goal to get outputs, outputs are generated by design’s model simulation, process of Error Localization is started immediately after process of parsing of design into model, and during process of simulation of model outputs for Verification and information for Model-Based Error Localization are obtained simultaneously.

b) implementation is more reasonable as both Error Localization and Correction require simulation of the model, and outputs, obtained during simulation for Error Localization and Correction can be used for Verification. From other hand, in a) implementation original design is used to get outputs, not parsed representation, that is more reliable. But if design is parsed correctly, then outputs of model simulation and design simulation should be same. In actual implementations of tool back-ends any of proposed algorithms can be used.

1. (a)basic and (b)more accurate Verification, Error Localisation and Error Correction implementations.

# Model for the C design

In order to be able to verify the design and further to localize and to correct an error in the design, the design should be parsed into the model. For the C design’s processing the model should be able to store following C design components:

* Flow control primitives (for, if, while, switch, do…while).
* Arithmetical operations
* Assignments
* User-defined and compound types and variables
* Pointer data types
* Pre-defined *C* functions

Any flow control primitive of the design can be parsed to conditions, operations, and transactions between them, and, therefore, minimally only two types of nodes are required to be implemented in the design's model - OperationNode, that has one predecessor and one successor and ConditionNode, that has one predecessor and two successors - true and false. Any additional information of the design can be stored in the separate structures of the hammock graph that is the hierarchy of the AstNodes of the corresponding hammock graph node, abstract syntax tree nodes.

The proposed FORENSIC model structure is single-entry single-exit structure, known as hammock graph [2][3], and is defined as follows.

Definition 1: *Hammock graph* is a structure *H=<N, E, n0, ne>*, where *N* is a set of nodes, *E* is a set of edges in *N×N*, *n0* is initial node and *ne* 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 *n0* to node *ne* is list of the nodes *p0, p1, ..., pk* such that *p0 = n0, p1 = n1, …, pk = ne*, and for all *i*, 0 ≤ *i* ≤ *k* – 1, (*pi, pi+1*) is in *E*. There is a path from *n0* to all other nodes in *N*. From all nodes in *N*, excluding *ne*, there is a path to *ne*.

Hammock graph is a special case of the flow graph when flow graph has one end node.

Nodes N of the hammock graph corresponds to the statements or part of the statements of the original design. Proposed structure is suitable for the Model-Based Error Localization algorithm's implementation, but for the Dynamic Slicing and Mutation-Based Error Correction algorithm's implementation more accurate structure is required, and, therefore, nodes of the hammock graph should be parsed further into abstract syntax tree representation, that is a hierarchy of operators, functions, and variables, and defined below.Definition 2: *Undirected graph* is a structure *U=<N, E>*, where *N* is a set of nodes, *E* is a set of simple edges in *N×N*. There exists a simple path from arbitrary node *n* in *N* to all other nodes in *N*. Path is a list of the nodes *p0, p1, ..., pk*such that *p0 = ni*, *p1 = ni+1*, …, *pk = nk*, and for all *i*, 0 ≤ *i* ≤ *k – 1*, (*pi, pi+1*) is in *E*.

Definition 3: *Abstract syntax tree* *A=<N, E,* n0*>* is a rooted undirected graph *U=<N, E\*>* with root n0, where two nodes are connected exactly by one simple path.

For the FORENSIC tool [1] object-oriented structure of the hammock graph and abstract syntax trees is implemented using C++ programming language, model was designed, developed and implemented in Graz University of Technology by Robert Könighofer and colleagues from the Institute for Applied Information Processing and Communications (IAIK) [7].

An example of the C design, parsed into the model, is Figure 2, for implementation details please refer to [1][7]. Siemens Corporate Research C design [9] is used for illustration in Figure 2.

l; 51 c:7 bb:2

schedule2/versions/v2/schedule2.c

argc!=4

UNEQUAL

Op

type: int

raw value: -1

Number

parsed value: -1

l; 51 c:38 bb:3

schedule2/versions/v2/schedule2.c

exit\_here(-1);

Y

l; 52 c:14 bb:4

schedule2/versions/v2/schedule2.c

prio=3;

N

l; 52 c:30 bb:13

schedule2/versions/v2/schedule2.c

prio.28=prio;

l; 52 c:5 bb:13

schedule2/versions/v2/schedule2.c

*prio.28>0*

l; 54 c:37 bb:5

schedule2/versions/v2/schedule2.c

prio.24=prio;

l; 52 c:30 bb:13

schedule2/versions/v2/sche

status=get\_command(&command

Y

N

FUN\_CALL

Op

argc

Id

argc\_2

ASSIGN

Op

type: int

raw value: 3

Number

parsed value: 3

ASSIGN

Op

AstNode

OperationNode

ConditionNode

1. An example of the FORENSIC tool design’s model. OperationNodes and ConditionNodes are hammock graph nodes, AstNodes are abstract syntax tree nodes.

# Model-Based Error Localization

Within contents of the DIAMOND FP7 project various error localization algorithms were investigated and implemented.

SMT-Based Error Localization and Correction algorithms were implemented in the Graz University of Technology. For the Symbolic Execution algorithm, the model was converted into the SMT-solver compatible expression and expression were solved for the unknown variables in the model. In order to execute algorithm additional build-in in the design verification in form of assert() statements are required, and, using the algorithm, erroneous assignment value of a variable in the design can be corrected. Concolic Execution does not require any built-in instrumentation, verifies design using simulation-based verification principles, but is time-consuming. SMT-Based algorithm's major disadvantage is that they correct errors in the integer type variables only, neither in floating point or compound types nor in operators.[7]

On the other hand, often much simpler errors appear in the C designs, like misuse of arithmetical operators or errors in one digit of the number. To process those types of errors it is possible to apply Model-Based Error Localization with Mutation-Based Error Correction, that is more specific, simpler and therefore faster than SMT-Based ones.

## Specification for the C design

In order to implement Model-Based Error Localization and Mutation-Based Error Correction algorithms, it necessary to verify the design, and for that, the specification, that defines the correct behavior of the design, should be provided.

Specifications can be formal or informal. In software industry informal specifications, that present design's entire structure using graphical modeling were applied at least since the late 70s, and, in parallel, formal specification languages, such as Larch, VDM, Z, FOOPS, and OBJ had been developed [4]. Formal specifications can be divided into two categories: explicit and non-explicit. Specifications, that is possible to animate, when outputs can be derived from the specification's inputs, are called explicit[5]. Non-explicit specifications are impossible to animate, and they are not used for error localization and error correction.

Various specification languages were investigated with the goal to be used as a specification language for the FORENSIC tool [1] when using the corresponding specification should be possible to explicitly define any possible C design's behavior and structure. The only solution, that is found, is to use the same specification language, as the design language. This type of specification can be seen as corresponding design's use cases or unit tests. Furthermore, specifications in the C format are simple to create and control, C code is executable, fast and reliable for animation purposes. Specifications using programming languages like Java, C, C++ are widely used in software industry [6].

Correspondence of the reference outputs, produced by the specification’s animation, and design’s outputs is reached by inserting so-called Observation Pointsinto the same locations of the specification and of the processing design.

## Simulation for the Model-Based Error Localization

If the specification for the design exists, reference outputs are obtained and design is parsed into the model, then it is possible to verify the design, for that design's model simulation should be implemented. During Simulation-Based Verification reference outputs of the design are compared with design's outputs, and the decision is made if design corresponds to its specification [6]. Later on, if design's verification fails, Model-Based Error Localization algorithm's implementation uses simulation trace of the model and Mutation-Based Error Correction algorithm's implementation verifies the corrected model.

Basically, it is possible to simulate the design’s model using two different approaches:

* Using direct simulation, when nodes of the hammock graph are executed from start node to the end node as they are activated, and intermediate simulation values are stored during simulation.
* By dumping model into *C* executable, instrumenting it to be able to obtain any possible intermediate values, and executing dumped code using *C* compiler, *gcc* as an example.

If the direct simulation is used, then activated C operators should be implemented and executed and values of the activated variables should be stored during processing. In order to store values of various C variable types, like pointers, arrays, compound data types etc., practically the memory model of the C compiler should be re-implemented, and this reimplementation is a duplication of the C functionality.

The much more natural approach is to use existing C functionality for simulation, when the model is dumped into C executable file, instrumented, executed, and output values are written into output files. This approach is faster and simpler than direct simulation.

In the FORENSIC tool [1] simulation of the model is implemented using C compiler. Furthermore, Dynamic Slicing algorithm that is implemented in the FORENSIC tool [1] using unique C variable’s addresses is implemented using C functionality for simulation.

Same principles for design's model simulation can be applied if the design is written using other programming languages, C/C++/SystemC or Java as examples.

## Model-Based Error Localization

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

Definition 4: *Activated path Ps* of the hammock graph *H=<N, E, n0, ne>* with input *s* *∈* *S* from node *n0* to node *ne* is a list of the nodes *n0, n1, ..., ne*, such that for all *j*, 0 ≤ *j* ≤ *e – 1*, (*nj, nj+1*) is in *E* and is activated with input *s* *∈* *S*.

Definition 5: *Activated nodes Na, n* ∈ *Na ∈ N* during simulation with input *s* *∈* *S* are nodes, that belong to the activated path *Ps* from node *n0* to node *ne* of the hammock graph *H=<N, E, n0, ne>.*

*Algorithm 1: Model-Based Error Localization.* During the Model-Based Error Localization algorithm's execution the model is simulated with inputs *S* and for each input *s ∈ S* output of the simulation is compared with reference output of the specification. If comparison, i.e. verification, fails, then activated nodes *n∈ Na* *∈ N* of the *hammock graph* *H=<N*, *E*, *n*0, *ne>* have *failed* counter increased, otherwise nodes have *passed* counter increased. During simulation with all inputs *S* set of total activated nodes *Nt* is cumulated by applying union of *Na* for each input *s ∈ S*, and after ranking algorithms are applied to the counters.

The list of Candidates for Correction *Cs*, that is a sorted list of the activated with all inputs *S* nodes *Nt* - *<N\*>* and output of an algorithm, is calculated for all ranking algorithms, except consecutive, as follows:

Let *C* be pair of sets of the nodes and the ranks, *C=<Nt, R>,* where *R* is the set of the rank values of the nodes *Nt*, then *Cs* are sorted by the rank value set of first element of the pair of the sets of nodes and the ranks *C*, .

The list of Candidates for Correction *Cs* for consecutive ranking algorithm is calculated as follows:

Let *C* be triplet of sets of the nodes, primary and secondary ranks, *C=<Nt, R1, R2>,* where *R1* is the set of the primary rank values of the nodes *Nt, R2* is the set of the secondary rank values. Let *C\** be initially sorted by the primary rank value triplet of the first, second and the third element of the triplet of the sets of the nodes, primary and secondary ranks *C*, , and let *C\*=<Nt\*, R1\*, R2\*>.* Later on, sorting using secondary ranking is applied to the nodes, that have same primary ranking value. During this second sorting *Cs* is cumulated using operation of addition to the end of the already sorted list (push):

*,*

*,*

*,*

,

,

.

Ranking algorithms, that are used in the FORENSIC tool’s [1] Model-Based Error Localization implementation are defined in [8][11][12] and in current manuscript and are following:

|  |
| --- |
| **Ample ranking** |
|  |
| **Consecutive ranking** |
| *rank1(n)* - simple ranking, *rank2(n)* - ochiai ranking |
| **Jaccard ranking** |
|  |
| **Ochiai ranking** |
|  |
| **Simple ranking** |
|  |
| **Tarantula tool ranking** |
|  |

where *rank* of the activated node *n∈ Na* *∈ N* is calculated using *passed(n)* – times passed for node *n*, *failed(n)* – times failed for node *n*, *totalpassed* – total passed simulations with inputs *S* and *totalfailed* – total failed simulations with inputs *S*.

For consecutive ranking algorithm's implementation, any secondary ranking from the proposed ones can be used, Ochiai is shown as an example. Consecutive ranking algorithm corresponds to Heuristic II and III, defined in [12].

It is possible to improve the error localization accuracy if to apply Dynamic Slicing algorithm to the activated during Model-Based Error Localization's simulation with input *s ∈* *S* nodes *Na.*

## Dynamic Slicing for the Model-Based Error Localization

Dynamic Slicing is an algorithm, that is applied to the nodes *Na*, activated during Model-Based Error Localization's simulation with input *s ∈* *S*. The idea behind Dynamic Slicing algorithm is following: some amount of nodes are activated during Model-Based Error Localization’s simulation with input *s ∈* *S*, but do not have any influence on simulation output, as an example those nodes can correspond to constant declarations and assignments at the beginning of the code. Dynamic Slicing algorithm allows to discard those nodes from the list of activated nodes *Na.*

In order define algorithm of the Dynamic Slicing we need to introduce following definitions - defined *DEF(n)* and referenced *REF(n)* variables of the node *n* ∈ *Na* *∈ N* [2].

Definition 6: Let *V* be the set of variables, that are in design's model, represented by a *hammock graph* *H=<N*, *E*, *n*0, *ne>.* Then for every node *n* ∈ *Na* *∈ N*, two sets can be defined, each of them is in *V*: *REF(n) – referenced variables* -set of variables, whose values are used or referenced in *n*, and *DEF(n) - defined variable* - variable, where value is assigned to in *n* [2].

Definition 7: *Dynamic Slice* of *Na* *∈ N* of the *hammock graph* *H=<N*, *E*, *n*0, *ne>* of the simulation with input *s* *∈* *S* is set of nodes, that have influence on the simulation output*.* List of *Dynamic Variables - DV(n)* at the moment of processing of the node *n∈ Na ∈ Ps* during Dynamic Slicing algorithm's execution includes variables, that have influence on output of further simulation, on processing of the nodes (*mi+1, ..., me*) *∈ Na ∈ Ps* , where *mi=n , ..., me=ne*.

*Algorithm 2:* *Dynamic Slicing*. Let activated nodesduring simulation with input *s* *∈* *S* be *Na* *∈* *Ps*, let *n∈ Na*. Variables, that are in the list of dynamic variables - *DV(n),* are updated at the moment of processing of every node *n,* and therefore let *DV(n)* be the list of dynamic variables at the beginning of processing of the node *n,* let *DV\*(n)* be the list at the end. Algorithm's processing goes from end activated node *ne* to initial activated node *n0*, and *DV(ni-1)≡DV\*(ni),* wheree ≤ *i* ≤ *1*. At the beginning of algorithm's execution dynamic variable's list is empty, *DV(ne)=∅,* and at the end is empty too - *DV\*(n0)=∅.*

If *n* contains a *С* function or a condition, then referenced variables *REF(n)* of the node *n* are added (union) to the list of dynamic variables *DV(n)*: *DV\*(n) = DV(n) ∪ REF(n)*. Output Observation Points or *assert()* statements can be last executed components - *ne* - and in this case output variables should be initially stored in the list of dynamic variables *DV\*( ne)*.

If node *n* contains an assignment, then referenced variables *REF(n)* of the assignment are added (union) to the list of dynamic variables *DV(n)* if and only if defined variable *DEF(n)* of the assignment is already in *DV(n)*, otherwise node *n* is *not in the dynamic slice* and the list of dynamic variables is propagated further, *DV\*(n)= DV(n)*. If *DEF(n)* is in *DV(n)* then defined variable *DEF(n)* should be removed from *DV(n)* before *REF(n)* addition: iff *DEF(n) ∈ DV(n)* then *DV\*(n) = (DV(n)* \ *DEF(n)) ∪ REF(n)*, otherwise *DV\*(n) = DV(n)* and *n* is *not in the dynamic slice*.

Nodes, that are *not in the dynamic slice* can be removed from *Na* *∈* *Ps* without change of the output of simulation, refer to Figure 3.

In the actual implementation of the Dynamic Slicing algorithm in the FORENSIC tool [1] unique C variable addresses are used instead of variable names, how it is shown in Figure 3. This makes possible to apply the Dynamic Slicing algorithm to any type of the design, where any combination of compound types and arrays is used as a variables.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Design | Activated Nodes *n*∈*Na* | *DEF(n)* | *REF(n)* | *DV\*(n)* | 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 |

1. An example of the Dynamic Slicing algorithm's execution, assignment at Line 6 is not in the dynamic slice.

# Mutation-Based Error Correction

When the error localization is complete and the list of the Candidates for Correction is generated, it is possible to correct an error in the design. Error classes define types of the errors, that can be found in the design, and valid error correction algorithm should be able to correct as much as possible of them.

If there is one error assumption for the error correction then mutations are applied one-by-one to the Candidates for Correction, if multiple error assumption, then mutations should be applied to several candidates simultaneously.

*Algorithm 3:* *Mutation-Based Error Correction*. Let the list of Candidates for Correction be *Cs=<N\*>,* where *N\** is set of sorted activated with all inputs *S* nodes *Nt* of the *hammock graph* *H=<N*, *E*, *n*0, *ne>,* let *n∈ N\*.* Let *O(n)* be arithmetical and logical operator(s) in *n* in *N\*, I(n)* be integer variable(s) in *n* in *N\**, *V(n)* be basic type variable(s) in *n* in *N\** (include *I(n)*), *Ct(n)* be number constants in *n* in *N\*, DP(n)* – Double-Precision numbers in *n* in *N\*, F() -* functions in *H*. Then

// Operators Mutations

is substituted with syntactically same one;

new model's Verification passes ? Error Is Corrected,

infinite loop during Verification ? break;

is restored,

Error Is Corrected ? End.

// Integer Variable's Mutations

is substituted with ,

new model's Verification passes ? Error Is Corrected,

infinite loop during Verification ? break;

is restored,

Error Is Corrected ? End.

// Basic Type Variable's Mutations

is substituted with *F()* if return type is same as at

new model's Verification passes ? Error Is Corrected,

infinite loop during Verification ? break;

is restored,

Error Is Corrected ? End.

// General constants mutations

is substituted with

new model's Verification passes ? Error Is Corrected,

infinite loop during Verification ? break;

is restored,

Error Is Corrected ? End.

// Mutations in constants digits

's every digit is substituted with values (0..9) and values (0..9) are added to the beginning of ,

new model's Verification passes ? Error Is Corrected,

infinite loop during Verification ? break;

is restored,

Error Is Corrected ? End.

// Double-Precision numbers rounding mutations

is rounded up and down,

new model's Verification passes ? Error Is Corrected,

infinite loop during Verification ? break;

is restored,

Error Is Corrected ? End.

No Correction Found.

# Experimental Results

Siemens Designs, that are used as input designs for experiments, are open-source C designs, created by Siemens Corporate Research and further extended by Mary Jean Harrold and Gregg Rothermel from Georgia Institute of Technology, Atlanta, USA [9].

It is proposed in [11] to measure the error localization accuracy using a number of statements, that is necessary to process before the error location is reached, but every statement of the design has various complexity. More accurate is to use number of mutations, required to correct the error as a meter, number of mutations does not depend on design entire structure, and this meter is used to measure the error localization accuracy in the FORENSIC tool [1]'s experimental results, in Tables I, III is mean number of mutations required to apply, in Table II is percentage, mean number divided by maximum possible number of mutations, in percents.

## Dynamic Slicing for the Model-Based Error Localization

The FORENSIC tool’s [1] experimental results, where the error localization accuracy with and without the Dynamic Slicing algorithm's use is compared when simple ranking for the Model-Based Error Localization algorithm is used, are in Table I.

1. Influence of Dynamic Slicing on error localization.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Design** | **% of corrected designs** | **Mean number of mutations required to be applied to correct the error when error localization is** | | **Change,** |
| **with Dynamic Slicing (A)** | **without Dynamic Slicing (B)** |
| print\_tokens | 28,57 | 1954,5 | 2106,0 | 7,19 |
| print\_tokens2 | 70,00 | 2141,14 | 2254,29 | 5,02 |
| replace | 50,00 | 1635,88 | 1683,06 | 2,8 |
| schedule | 33,3 | 564,0 | 715,33 | 21,16 |
| schedule2 | 40,00 | 648,25 | 713,25 | 9,11 |
| tcas | 75,61 | 304,0 | 553,71 | 45,10 |
| tot\_info | 56,25 | 1338,5 | 1363,94 | 1,87 |

Influence of the Dynamic Slicing algorithm on the error localization accuracy is defined by design’s internal structure: if the design has a number of initializations at the beginning of the code, then the Dynamic Slicing algorithm can discard a number of irrelevant initializations. If Dynamic Slicing algorithm is used then error localization accuracy is strongly increased for *tcas* design from the Siemens Designs - by 45,1%, for *schedule -* by 21,16% and *schedule2* - by 9,11%, but less for *print\_tokens*, *print\_tokens2*, *replace* and *tot\_info* designs.

## Model-Based Error Localizsation using various rankings

Experimental results, where accuracy of the Model-Based Error Localization algorithm using various rankings, implemented in the FORENSIC tool [1] is compared, are in Table II and corresponding graphical representation is Figure 4. The Dynamic Slicing algorithm is used, and this algorithm's configuration's time requirements are comparable with the Mutation-Based Error Correction algorithm's time requirements, that are shown in Table III.

1. Model-Based Error Localization using various rankings – % of mutations required to correct design.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Rank**  **Design** | **Ample** | **Consecutive** | **Jaccard** | **Ochiai** | **Simple** | **Tarantula** |
| print\_tokens | 12,34 | 8,50 | 9,05 | 9,05 | 20,22 | 9,05 |
| print\_tokens2 | 4,29 | 1,34 | 10,88 | 2,65 | 25,61 | 12,90 |
| replace | 5,93 | 3,0 | 5,82 | 4,19 | 12,23 | 5,97 |
| schedule | 5,15 | 2,55 | 6,88 | 2,63 | 17,53 | 7,09 |
| schedule2 | 50,04 | 37,49 | 45,22 | 41,92 | 15,33 | 45,22 |
| tcas | 8,40 | 5,99 | 9,51 | 6,72 | 8,04 | 9,86 |
| tot\_info | 29,16 | 8,03 | 19,67 | 9,81 | 16.62 | 21,57 |
| std. deviation | 16,14 | 12,51 | 13,13 | 13,69 | 5,81 | 12,95 |
| weighted mean\* | 14,20 | 6,94 | 12,81 | 8,20 | 13,30 | 13,58 |

\*Number of the designs in design's group is Table IV(C).

1. The Model-Based Error Localization algorithm's implementation using various rankings and the Dynamic Slucing algorithm use – % of mutations required to correct the error in the design.

Simple ranking for the error localization appears to be more stable than other rankings, it's results have lowest standard deviation. Error localization implementation using consecutive ranking is most accurate from investigated ones, it is 1,26% more accurate than next Ochiai ranking and 7,26% more accurate than worst ample ranking.

## Mutation-Based Error Correction

A number of corrected by the Mutation-Based Error Correction algorithm designs strongly depends on the applied by the algorithm mutation types, and current algorithm's version's experimental results with algorithm's time requirements are in Table III. One error assumption, simple ranking for the Model-Based Error Localization algorithm and the Dynamic Slicing algorithm are used.

1. Mutation-Based Error Correction

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Design** | **Complexity, lines** | **% of corrected designs** | **Mean # of mutations required to correct design** | **Time requirements\*\*, s** |
| print\_tokens | 569 | 2/7 - 28,57 | 1954,5 | 507 |
| print\_tokens2 | 515 | 7/10 - 70,00 | 2124,14 | 786 |
| replace | 558 | 16/32 - 50,00 | 1635,88 | 545 |
| schedule | 419 | 3/9 - 33,33 | 564 | 204 |
| schedule2 | 312 | 4/10 - 40,00 | 648,25 | 286 |
| tcas | 186 | 31/41 - 75,61 | 304 | 84 |
| tot\_info | 413 | 18/32 - 56,25 | 1338,5 | 475 |

\*\* 1000 inputs for the algorithm's verification is used.

In [10] similar error correction experiments are shown when Tarantula tool ranking was used for error localization's ranking, and results are compared in the table below.

1. Number of errors corrected in [10] comparing with the FORENSIC Mutation-Based Error Correction

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Design** | **# of corrected designs in [10] (A)** | **# of corrected designs using *FORENSIC* tool(B)** | **Total # of designs (C)** | **Difference,** |
| print\_tokens | 0 | 2 | 7 | 28,75 |
| print\_tokens2 | 0 | 7 | 10 | 70,00 |
| replace | 3 | 16 | 32 | 40,63 |
| schedule | 0 | 3 | 9 | 33,33 |
| schedule2 | 1 | 4 | 10 | 30,00 |
| tcas | 9 | 31 | 41 | 53,66 |
| tot\_info | 8 | 18 | 23 | 43,48 |
| % corrected | 15,9 | 61,36 |  | 45,45 |

Mutation-Based Error Correction algorithm of the FORENSIC tool [1] corrects *61.36%* of the Siemens Designs [9], that is *45.45%* more than in [10], this is because in [10] error correction algorithm applies mutations in *C* operators only, but FORENSIC tool's [1] Mutation-Based Error Correction algorithm applies more various mutations.

**Note.** Research is implemented within contents of FP7-ICT-2009-4-248613 DIAMOND project.

##### References

1. FORENSIC tool, http://www.informatik.uni-bremen.de/agra/eng/forensic.php, 2012.
2. Weiser, Mark, Program Slicing, Software Engineering, IEEE Transactions on, pp. 352–357, 1984.
3. Kas'janov, V. N., Distinguishing Hammocks in a Directed Graph, Soviet Math. Doklady, vol. 16, no. 5, pp. 448-450, 1975.
4. Jadish S. Gangolly, Lecture notes on Design & Analysis of Accounting Information Systems, University at Albany, New York, 2000.
5. Xiaoping Jia, An approach to animating Z specifications, Computer Software and Applications Conference, 1995, pp. 108 – 113, 1995.
6. William K. Lam, Hardware Design Verification. Simulation and Formal Method- Based Approaches, 2005.
7. Georg Hofferek, Robert Könighofer, Jörn-Marc Schmidt, Görschwin Fey, Alexander Finder, Andre Sülflow, Urban Ingelsson, Urmas Repinski, Jaan Raik, Stephen Scholefield, FoREnSiC - An Automatic Debugging Environment for C Programs, Haifa Verification Conference, pp. 260-265, 2012
8. Urmas Repinski and Jaan Raik, Simulation-Based Verification with Model-Based Diagnosis of C Designs, East-West Design Test Symposium. pp. 42-45, 2012
9. Bug Aristotle Analysis System -- Siemens Programs, HR Variants, http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/, 2012
10. Vidroha Debroy and W. Eric Wong, Using Mutation to Automatically Suggest Fixes for Faulty Programs, Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, pp. 65-74. 2003
11. 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.
12. Quek, C. and Leitch, R.R., Using energy constraints to generate fault candidates in model based diagnosis, Intelligent Systems Engineering, First International Conference on (Conf. Publ. No. 360), pp.159 – 164, 1992.
13. GIMPLE plug-in, http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html, 2012.