ронные сети, генетические алгоритмы и нечеткие системы. М.: Горячая линия-Телеком, 2008. 452 с.

References

1. Ayvazyan S.A., Bukhshtaber V.M., Enyukov I.S., Meshalkin L.D., *Prikladnaya statistika: Klassifikatsiya i snizhenie razmernosti* [Applied statistics: Classification and reduction of dimensionality], Moscow, 1989.

2. Ayzerman M.A., Bravermann Je.M., Rozonoer L.I., *Metod potentsyalnykh funktsy v teorii obucheniya mashin* [Method of potential functions in the theory of machine learning], Moscow, Nauka, 1970.

3. Vapnik V.N., Chervonenkis A.Y., *Teoriya raspoznavaniya obrazov (statisticheskie problemy obucheniya)* [Recognition theory (statistical learning problems)], Moscow, Nauka, 1974.

4. Zhuravlev Y.I., *Izbrannye nauchnye trudy* [Favorites scientific works], Moscow, 1998, 420 p.

5. Zagoruyko N.G., *Prikladnye metody analiza dannykh i znany* [Applied methods of data analysis and knowledge], Novosibirsk, 1999, 270 p.

6. Mazurov V.D., *Metod komitetov v zadachakh optimizatsii i klassifikatsii* [Committee method in optimization and classification], Moscow, Nauka, 1990, 248 p.

7. Rastrigin L.A., Erenshteyn R.H., *Metod kollektivnogo raspoznavaniya* [Method of collective recognition], Moscow, 1981.

8. Rifkin R., Klautau A., *Journ. of Machine Learning Research*, 2004, pp. 101–141.

9. Freitas A.A., *Data Mining and Knowledge Discovery with Evolutionary Algorithms,* Berlin, Springer, 2002, 279 p.

10. Emelyanov V.V., Kureychik V.V., Kureychik V.M., *Teoriya i praktika evolyutsionnogo modelirovaniya* [Theory and practice of evolutionary modeling.], Moscow, 2003.

11. Rutkovskya D., Pilinsky M., Rutkovsky L., *Nejronnye seti, geneticheskie algoritmy i nechetkie sistemy* [Neural networks, genetic algorithms and fuzzy systems], Moscow, 2008, 452 p.

UDS 62

MODEL-BASED VERIFICATON WITH ERROR LOCALIZATION AND ERROR CORRECTION FOR C DESIGNS

(The work has been partly supported by European Commission FP7-ICT-2009-4-248613 DIAMOND project, by Research Centre CEBE funded by European Union through the European Structural Funds, by Estonian Science Foundation grants 9429 and 8478 and by Estonian Academy of Security Sciences)

Urmas Repinski, Assistant
(Department of Computer Engineering Tallinn University of Technology
5, Ehitajate tee, Tallinn, 19086, Estonia, urrimus@hotmail.com)

Abstract. Process of software design verification makes sure if design holds its specification, existence of the specification that is possible to animate allows to perform simulation-based verification. In order to locate and repair errors if verification fails, we have to have access to the structure of the debugged design. For this purpose the design should be parsed into suitable representation - into the model. Both algorithms: Error Localization process and Error Correction process of the design require model simulation. This article presents different simulation algorithms. Simulation of the model is usually applied directly, when the design model is simulated with the goal to obtain the outputs from the inputs, but this approach is not always suitable, because in this case the functionality of the programming language design should be almost completely re-implemented for simulation. An alternative approach described in this article – design model simulation using design programming language functionality. It is more reasonable and does not require re-implementation of the functionality already available in design programming language. This approach also makes possible implementation of the algorithm of dynamic slicing for Error Localization and Error Correction.

Keywords: simulation-based verification; error localization; error correction, specification; automatic error correction, debug; C design.

In simulation-based verification, a digital system (design) looks like a *black box*, as it uses inputs and compares outputs. It's entire design structure is irrelevant. If we want to debug this design, it should be transformed into a *white box* structure. 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 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.

This paper presents implementation of the *Model-Based Error Localization* algorithm for *C* designs. The algorithm has been integrated to the open source FoREnSiC [1] (FOrmal Repair ENvironment for SImple C) tool. The *FORENSIC* model has a flow-chart-like structure that can be described as a special case of flowgraph: the *hammock graph* [2]. The tool includes a *Front-End* parser, a number of *Back-End Error Localization* and *Error Correction* algorithms' implementations.

Similar to [3], 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 devia-

tion from exact localization.

Model-Based Error Localization algorithms store trace of model simulation and select most probable erroneous components of the model. Dynamic Slicing can increase Error Localization accuracy that will be shown using experimental results, obtained by FORENSIC Error Localization tool. Level of design details in the model should be high enough to be able to extract variables, variable names, operators and conditions for Dynamic Slicing.

The article will show what structure is possible for the model for Error Localization, that were implemented in FORENSIC tool [1], how functionalities, provided by the model, allow to implement Model-Based Error Localization, Mutation-Based Error Correction and Dynamic Slicing for C designs, how parser for the model can be implemented using C/C++ GIMPLE plugin. FORENSIC tool [1] can be extended to C/C++/SystemC design's support in the future.

FORENSIC tool structure

FORENSIC tool[1] is divided into front-end and back-end components. Front-end parses design into the model and back-end consists of several components that are responsible for different functionality, such as design Error Localization and Error Correction.

Front-End Implementation. Before starting process of Verification with Error Localization with Dynamic Slicing, based on the model, it is necessary to parse processed design into the model representation.

GIMPLE plug-in for GCC compiler [4] 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 parsing, as it has whole information about processed design. At this stage there is no need to create additional parser or lexical translator, C compiler's plug-ins already generate suitable design form, and just necessary to store the parsed design tree into the model. For that every class of the model implementation should have add and remove methods.

The front-end parser of the FORENSIC tool [1] for C design was implemented at University of Bremen. The front-end of the tool can be extended to *C++* and to *SystemC* support for C++ and SystemC language's support.

Tool Implementation. Various back-ends and algorithms are implemented in FORENSIC tool [1] for Error Localization and Correction, Simulation-Based back-end, 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, places where data information is stored during processing. It includes model with the most significant data structure in tool;

– Functionality, is front-end, back-end, animation and simulation algorithms implementations, are actual implementations that were developed and documented and can be extended further.

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.

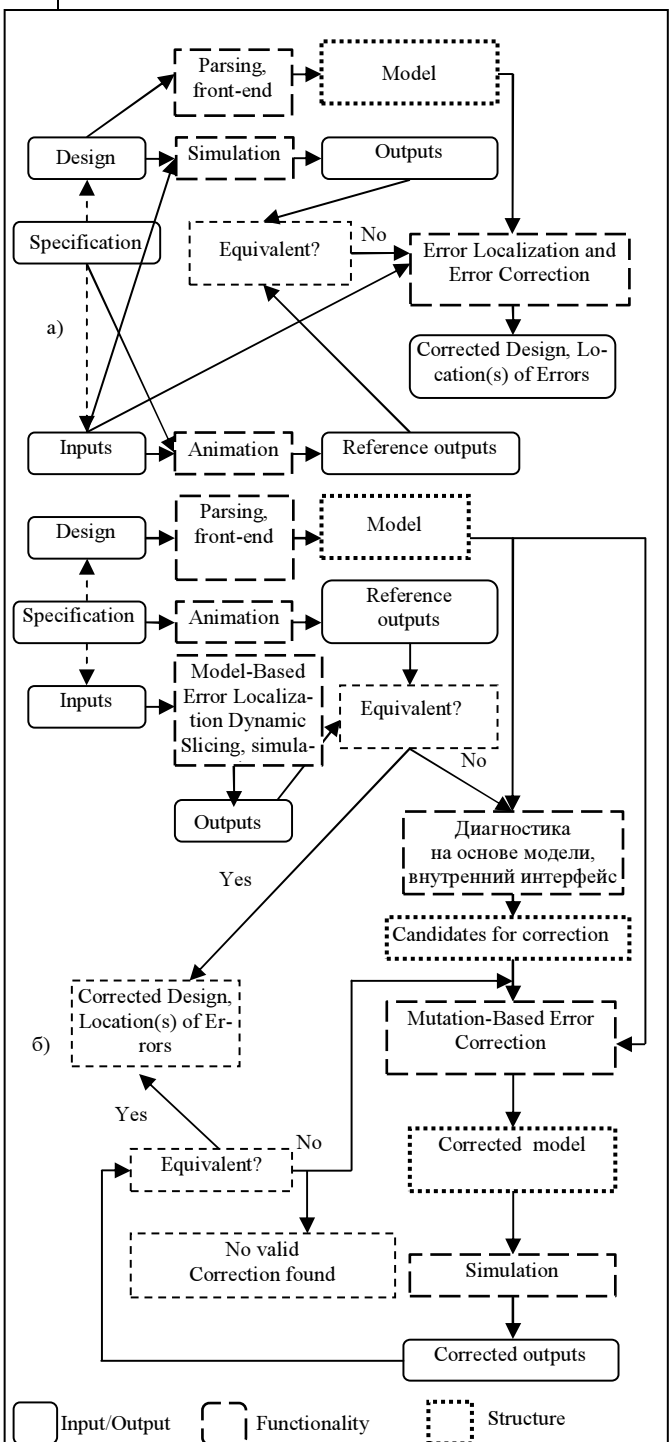

Figure. 1. Basic (a) and more (b) accurate Verification, Error Localisation and Error Correction implementations

The 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. 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. On the 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 the same. In actual implementations of tool back-ends can be used any of the proposed algorithms.

Further Model-Based Error Localization and Mutation-Based Error Correction functionalities of FORENSIC tool [1], simulation functionality and structure of specification for the Error Localization and Error Correction are described in details.

Model for Error Localization

Model Difinition. In order to perform design Verification and further Error Localization, a model, that is suitable representation for Error Localization of the processed design, should be developed. Design can be presented as single-entry single-exist structure [5], that is a special case of *flow graph* [6], known as *hammock graph* [6]. 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 proposed model structure nodes in the *hammock graph* are statements or part of statements of the original design, that include assignments, conditions, arithmetical operations, etc. This model structure is suitable for Error Localization implementation, but for Dynamic Slicing and Mutation-Based Error Correction model representation should be more accurate. It is necessary to have possibility to obtain statement's variables in the node and to process defined and referenced variables in statement for Dynamic Slicing algorithm implementation, that will be shown further. For this purposes every node in hammock graph should be parsed further into *flow graph* representation, that is hierarchy of operators and functions with corresponding variables in the statement.

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.

Definition 2. *Flow graph* is a structure $H=<N, E, n_0>$, where N is a set of nodes, E is a set of edges in $N\times N$, n_0 is an initial node. If (n, m) is in E then n is an immediate predecessor of m and m is an immediate successor of n. There exists a path from n_0 to all other nodes in N. Path is a list of nodes $p_0, p_1, ..., p_k$ such that $p_0=n_0, p=n_1, \ldots, p_k=n_k$, and for all i, $1\le i\le k-1$, (p_i, p_{i+1}) is in E.

Practically in FORENSIC tool corresponding structure is implemented using C++ classes, every *hammock graph* node and *flow graph* node has number of class variables and functions that allow to store design as *"white box"* representation [1]. Model were designed and implemented in Graz University of Technology.

Implementation of C model. The model should store all possible information about the design and allow to trace hammock graph nodes during simulation, to obtain defined and referenced variables during simulation, to correct and change any part of the processed design.

The model built for C designs 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.

Proposed model is implemented using C++ classes, C++ programming languages, in FORENSIC tool [1].

For this purposes model consists of *FunctionData* class instances, that has number of parameters, that can be *CompoundType* or *BasicType* class instances. Compound Data Type has number of methods and class variables, that allow to define user-defined C data type. *BasicType* defines pointers and pre-defined C data types.

Every data type can be defined using pointer, address in memory where data type is located. For each type number of pointer indirections parameter exists in the model. If this value is zero, then we have non-pointer data type, if greater than zero then have pointer type, pointer to pointer type, etc.

Every function consists of the start and end node, between them is model's hammock graph (def. 1), where processed design statements are stored. Hammock graph nodes are implemented in FORENSIC

tool [1] by *FlowGraphNode* class instances. *FlowGraphNode* class can be extended by *ConditionNode* class, that has one predecessor and two successors – if condition at the *ConditionNode* is true or false, and *OperationNode*, that is statement with no condition, one predecessor and one successor. *FlowGraphNode* class instances are presented by *Flow Graph* (def. 2), *AstNode* class instances, that store parsed representation of assignments and arithmetical operations in the node. C functions are presented using *OperationNodes* with special description (a function), with function's return value and parameter's representation using *AstNode* class instances.

Described model is sufficient enough to store any C design, all C flow control primitives can be stored using this structure.

Additional nodes with comments and any additional information for processing can also be added to the model.

The example of FORENSIC tool [1] model is Figure 2.

Model-Based Error Localization

In Graz University of Technology SMT-Based Error Localization and Correction algorithm were implemented, when simulation trace of the model with specific inputs is transformed into SMT-solver compatible expression and expression is solved setting repaired variable as unknown variable.

This algorithm, named Symbolic Execution, is repeated for every processed variable in the model. Algorithm allows to correct complicated errors when variables or functions are defined erroneously.

However in practice, simpler errors such as misuse of arithmetical operator and error in one digit of the number appear much more often. To correct these types of errors Model-Based Error Localization with Mutation-Based Error Correction Algorithm can be applied. It is simpler and faster than more sophisticated algorithms. But if Model-Based Error Localization algorithm does not correct errors and verification of design still fails, then SMT-Based Error Localization and Correction algorithm can be applied after Model-Based Error Localization [11].

In FORENSIC tool [1] it is possible to execute back-ends independently, one after another.

Specification for C design. In order to implement Model-Based Error Localization, it is necessary to verify design. For that we need to have specification in valid format, that is *explicit* and possible to *animate*, that will produce reference outputs from inputs which will be compared further with design's outputs.

Specifications can be *formal* or *informal*. In information systems development in business *informal* specifications through graphical modeling have been used at least since late 70s. Recently formal specification languages (such as Larch, VDM, Z, FOOPS and OBJ) have been developed [8].

Formal Specifications are divided into two categories: *explicit* and *non-explicit*. Specifications that are possible to *animate,* called *explicit,* are defined in such a way when specification's outputs are derived from specification's inputs directly.[9] *Non-explicit* specifications are impossible to animate.

Difficulty of defining *C/C++* design's specification's format is that this should be specification suitable for any *C/C++* design. It should be possible to define any C/C++ design's behaviour using that. *C/C++* design has no functionality for temporal operators like *SystemC*; most of its functionality are arithmetical procedures, conditions and file processing.

Best solution to define specification of system

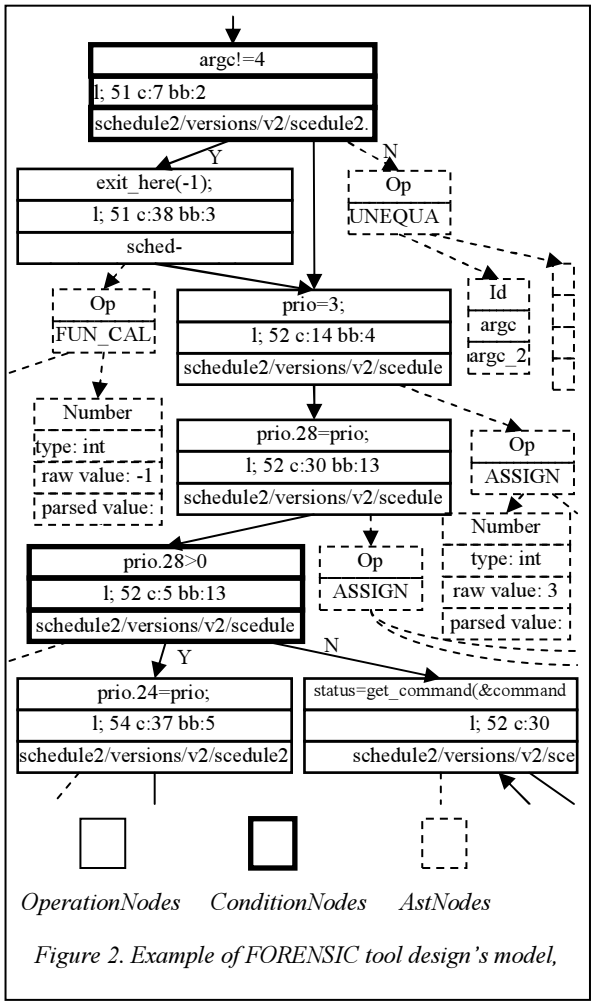

Figure 2. Example of FORENSIC tool design's model,

with required properties – specification using *C/C++* language, same language as design. Specifications in *C/C++* format are simple to create and to control. For verification purposes specification should be possible to *animate* and *C/C++* code is executable, fast and reliable for this purposes.

Specifications using high-level programming languages like *Java, C, C++* are widely used in software development industry [9] .

If specification for *C/C++* design is *C/C++* specification language, then it is easy to produce outputs in

the same format as in processed design. To do this *Observation Points* in the same locations for specification and for processed design can be added to both design and specification.

In this case internal format of specification is not strictly defined and any *C/C++* functionality can be used in specification. As *C/C++* is high-level programming language, specification can be implemented using test cases for arbitrary functions, or they can model design more precisely, most important that outputs of specification should correspond to outputs of design.

In general, it is a good idea to use same *design* language and *specification* language.

Simulation for Model-Based Error Localization. When specification for the design is ready, and design is parsed into model, it is possible to verify the design. During Simulation-Based Verification reference outputs of the design are compared with outputs, and if design corresponds to its specification decision is made. For obtaining the outputs from design inputs it is enough to execute the design. However, for Model-Based Error Localization we need trance of simulation of the design, storing information about components of the design that were activated during simulation. It is necessary to be able simulate design's model, and process of design verification and Error Localization can be merged together using simulation of the model only for Verification and Error Localization.

In the FORENSIC tool [1], simulation of the model is implemented using C functionality, executing instrumented and dumped model of the C design. The advantages of using C language's functionality for simulation of C design's model over direct simulation of the C design's model are as follows.

In case of direct simulation of the 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 different: pointers, arrays, compound data types and so on, and practically memory model of C programming language should be rewritten in order to store the values. Therefore using already existing C compiler's memory model is reasonable. Additionally, implementation of operators would be duplication of C functionality.

On the other hand, when using C functionality, the model should be dumped into a C executable file, instrumented, executed, and output responses should be written into output. It allows getting any data during simulation process, and we do not lose any functionality, but get benefits such as speed of execution and simplicity of approach. There are not any programming errors while complicated C functionality is re-implemented. *Dynamic Slicing* described in Section 2.4 is implemented in the *FORENSIC* tool [1] based on *C* functionality.

Same principles of design's model simulation can be applied if the design is written using on other programming language, C/C++/SystemC or JAVA as examples.

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

***Definition 3*:** 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 *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 4*:** Activated nodes $N_a \in N$ during simulation with input *s* that is in 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.* During Model-Based Error Localization the model is simulated with inputs S and for each input $s \in S$ output of simulation is compared with reference output of 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 inputs S ranking algorithms are applied to counters, and finally nodes are sorted according to their rank. Nodes with high ranks are *Candidates for Correction* and stored in corresponding data structure.

Ranking for Error Localization. During *Model-Based Error Localization* rankings are applied to model's *Activated Nodes*. It allows extracting *Candidates for Correction*. Ranking equations and comparison of rankings using Siemens Designs [10] were studied in [3]. 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 purpose of this work was not correcting errors as in [11], but comparing coefficients for ranking algorithms.

Coefficients described in this paper are taken from diagnosis/automated debugging tools Pinpoint [12], Tarantula [11], and AMPLE (Analyzing Method Patterns to Locate Errors) [8], and from molecular biology domain (Ochiai coefficient) [3].

A program spectrum [13] is collected in [3] of processed designs. Based on obtained spectra, that is *activated nodes* and *passed* and *failed* information in terminology introduced above, binary matrix $M \times N$ is built: M simulations of design of N blocks length [3].

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 was *passed*. Using notation of [14], 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

(1)

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

Table 1

Comparison of ranking algorithms for FORENSIC Simulation-Based Back-End with experiments in [3]

[9] similarity coefficients s_j	FORENSIC tool ranking
Tarantula tool ranking[7]	
$s_j = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$	$rank(n) = \frac{\frac{failed(n)}{totalfailed}}{\frac{passed(n)}{totalpassed}+\frac{failed(n)}{totalfailed}}$
Jaccard coefficient used in Pinpoint framework [9, 1]	
$s_j = \frac{a_{11}}{a_{11}+a_{01}+a_{10}}$	$rank(n) = \frac{failed(n)}{passed(n)+totalfailed}$
Ochiai coefficient, used in molecular biology [9]	
$s_j = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})\cdot(a_{11}+a_{10})}}$	$rank(n) = \frac{failed(n)}{\sqrt{totalfailed\cdot(passed(n)+failed(n))}}$
AMPLE tool ranking [9]	
$s_j = \left\lvert\frac{a_{11}}{a_{11}+a_{01}} - \frac{a_{10}}{a_{10}+a_{00}}\right\rvert$	$rank(n) = \left\lvert\frac{failed(n)}{totalfailed} - \frac{passed(n)}{totalpassed}\right\rvert$
***simple* ranking introduced in current article**	
	$rank(n) = \frac{failed(n)}{totalfailed}$
Counters a_{ij} are defined in (1)	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.

It is possible to improve Error Localization accuracy by applying additional dynamic slicing algorithm to the *Candidates for Correction*.

Dynamic Slicing for Model-Based Error Localization. *Dynamic Slicing* is a technique applied to *Activated Nodes* N_a obtained during *Model-Based Error Localisation* simulation that allows reducing number of *Candidates for Correction*. The idea behind *Dynamic Slicing* is the following: some amount of candidates (nodes) is 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 so on. *Dynamic Slicing* allows discarding those statements.

***Definition 5*:** *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. $N_a \backslash N_d$ can be removed from *Activated Path* of simulation with input *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 *FlowGraphNode* in *FORENSIC* implementation [1]).

***Definition 6*:** 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*. This 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 $gREF=\varnothing$.

If *n* contains *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 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 $gREF=(gREF \backslash DEF(n)) \cup REF(n)$, otherwise $gREF=gREF \backslash DEF(n)$ 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.

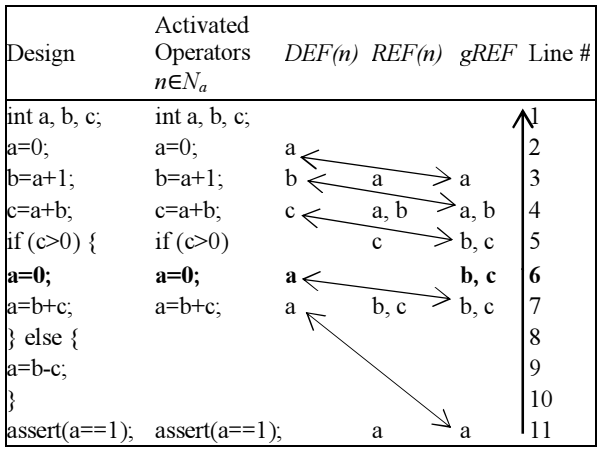

Figure 3. Example of Dynamic Slicing execution, assignment at Line 6 is not in slice

Depending of processed design's nature, *Dynamic*

Slicing can have strong influence. Number of assignments can be made at the initializing part of design, those are usually constants, arrays, and so on. 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 [1] unique *C* variable addresses are used instead of variable names, that are shown at *Figure 3*, that were possible to implement only because of *C* functionality were used for simulation of model, variables are parsed in model and presented using *Flow Graph*.

Mutation-Based Error Correction

When Error Localization is complete and Candidates for Correction are generated it is possible to correct error in design. Error Classes define types of errors that could be found in processed designs, and valid Error Correction algorithm should be able to correct as much as possible. The model with implemented Mutation-Based Error Correction (FORENSIC model) defines possible corrections implementation, and therefore model choice have high influence on correction procedure and results.

Error Classes found in experimental Siemens Designs[13] that are used for experiments with FORENSIC tool [1] are described below, last column if Mutation-Based Error Correction algorithm of FORENSIC tool [1] corrects corresponding class:

Table 2

Error Classes in Siemens Benchmarks

Error class	Operators/examples	Supported?
AOR	ADD (+), SUB (-), MULT (*), DIV (/), MOD (%)	YES
ROR	EQ (==), NEQ (!=), GT (>), LT(<), GE (>=), LE (<=)	YES
LCR	AND (&&), OR (||)	YES
ASOR	(+=), (-=), (=)	YES
Integer mutation	500 → 550, 740 → 700, 1 → 2, a[0] → a[1], n= 0 → n = 1	Partially
Float mutation	0.0 → 0.1, 0.0 → 1.0	Partially
Incorrect array index	a[i-1] → a[i], a[i+1] → a[i], a[3] → a[4], a[var] → a[0]	Partially
+1 added/missing	r = i → r = i+1, f(a, b, c+1) → f(a, b, c), if(i) → if(i+1)	Partially
Constant mutation	TRUE → FALSE	NO
Float rounded	2,5066282 → 2,50663 EPS → EPS-0.000001	Partially
Missing code/Code Added	/* && (Cur_Vertical_Sep > MAXALTDIFF); missing code */	NO
Function Substituted by Constant	Inhibit_Biased_Climb() → Up_Separation	NO
Logic Change	Brackets removed	Partially

Some of the Error Classes are not supported by current version of Mutation-Based Error Correction algorithm. Some of them are because of limitation of Mutation-Based Error Correction algorithm, as it does not support corrections of complicated errors. Others are impossible to correct when some code is simply missing in processed design.

Algorithm. Mutation-Based Error Correction algorithm corrects errors using following mutations.

- Operators Mutation – operators are mutated with similar ones. Operators are divided into several groups where similar operators are grouped together.
- General Mutations of constants. Those include adding one to constant - "C → C+1", subtracting one from the number - "C → C-1", setting constant to zero - "C → 0", and finally making constant negative "C → -C".
- Mutations in constant number's digits. Number's digit is mutated, values "0...9" are applied to every digit, numbers can be any C floats or integers.
- Double-Precision numbers rounding mutations. Double-precision numbers (floats and doubles) are mutated with rounded ones, rounded up and down.

Groups of operators for mutations in operators are.

- Arithmetical Operators: plus (+), minus (-), multiplication (*), division (/), module (%).
- Assignment Operators: assign (=), plus assignment (+=), minus assignment (-=), multiplication assignment (*=), division assignment (/=), module assignment (%=).
- Comparison Operators: less (<), grater (>), equal (==), unequal (!=), less equal (<=), grater equal (>=).
- Logical Operators: logical and (&&), logical or (||)
- Bit Shift Operators: bit shift left (<<), bit shift right (>>), bit and (&), bit or (|), bit xor (^).
- Bit Shift With Assignments Operators: bit shift left assignment (<<=), bit shift right assignment (>>=), bit and assignment (&=), bit or assignment (|=), bit or assignment (^=).
- Unary Arithmetical Operators: unary plus (+), unary minus (-).
- Increase Decrease Operators: pre increase (++x), post increase (x++), pre decrease (--x), post decrease (x--).
- Unary Logical Operators: logical not (!), bit complement (~).

If there is *one* fault assumption for Error Correction then mutations are applied one-by-one to Candidates for Correction, and mutated model is verified again using C compiler. However, if *multiple* fault assumption is used, then mutations should be applied to several candidates simultaneously to get valid correction. If verification passes, then valid correction is found, if differ, then process of Mutation-Based Error Correction is continued while all Candidates for Correction are processed (Figure 1b).

Algorithm 3: *Mutation-Based Error Correction.* Let *C* be *Candidates for correction*, $C=<N, R>$ where *n* in *N* is node of model – *hammock graph* $H=<N, E, n_0, n_e>$, *r* in *R* – rank of node, calculated during *Error Localization* using ranking algorithm. Let *O(n)* be arithmetical and logical operator(s) in *n* in *N*, *Ct(n)* be

constants in n in N, $DP(n)$ – Double-Precision numbers in n in N. Initially C is sorted according to rank, $C_s = \langle \binom{\max(R) \dots \min(R)}{N}, \binom{\max(R) \dots \min(R)}{R} \rangle$, and then
$\forall n_i \in N \in C_s, i = 1 \dots |N|$:
  $\forall O(n_i)$:
    $O(n_i)$ is substituted with one from the same group,
  new model's Verification passes ? Error Is Corrected,
  infinite loop during Verification ? break;
  $O(n_i)$ is restored,
  Error Is Corrected ? End.
  $\forall Ct(n_i)$:
  $Ct(n_i)$'s every digit is substituted with values (0..9),
  new model's Verification passes ? Error Is Corrected,
  infinite loop during Verification ? break;
  $Ct(n_i)$ is restored,
  Error Is Corrected ? End.
  $\forall DP(n_i)$:
  $DP(n_i)$ is rounded up and down,
  new model's Verification passes ? Error Is Corrected,
  infinite loop during Verification ? break;
  $DP(n_i)$ is restored,
  Error Is Corrected ? End.
No Correction Found.

Mutation-Based Error Correction algorithm does not correct complicated errors when some code is missing or function is substituted by constant, but has its own advantages. Error Classes supported by Mutation-Based Error Correction of FORENSIC tool [1] are natural and not complicated, algorithm is fast, reliable and simple. According to experimental results, processing time with Mutation-Based Error Correction algorithm of Siemens Benchmark [10] design, that have 180–570 lines of C code takes from 1 to 6 minutes with 2,4 GHz processor PC.

Property of Model-Based Error Correction. In order to measure Error Localization accuracy, in [14] number of statements to reach the error is used. But every statement of processed design can have different complexity, as examples it can be assignment, assignment with arithmetical operators, function with different number of parameters, every parameter of function can be other function or arithmetic, it can be simple logical condition or very complicated logical condition. In order to make code more readable and beautiful, sometimes it is a "good style" to compose complicated statements that are easier to observe for code developer than group of small statements. In this case approach to count number of statements that are necessary to reach error is not accurate measure of Error Localization accuracy.

On the other hand, number of mutations, that is applied to parsed design's model, should be practically the same for same functionality. This is because in FORENSIC model hammock graph nodes have no more than 3 operands, and complicated statements are presented by several hammock graph nodes. Implemented mutations for Mutation-Based Error Correction define operators, numbers, constants and logic, that is processed during Correction, or define locations of errors, that can be found and corrected in the design. Not processed during Mutation-Based Error Correction errors are errors, that are out of scope of investigation in the design, and can be omitted during measurement of Error Localization accuracy without losing measurement accuracy.

It means that counting number of processed mutations during Mutation-Based Error Correction instead of number of statements, required to reach the error, is more accurate meter of Error Localization accuracy.

Experimental Results

Experiments of Model-Based Error Localization with Mutation-Based Error Correction were performed using Error Localization with Dynamic Slicing and without Dynamic Slicing. Experimental designs are C Siemens Benchmarks [10].

Table 3

Siemens Benchmarks

Design	Complexity, lines	Number of Inputs	Number of Erroneous Designs
tcas	186	1605	41
schedule	419	2650	9
schedule2	312	2710	9
replace	558	5543	31
tot_info	413	1052	22
print_tokens	569	4130	7
print_tokens2	515	4115	10

Siemens Benchmarks [10] are suitable for FORENSIC tool [1] experiments, those are open-source C designs, created by Siemens Corporate Research and extended by Mary Jean Harrold and Gregg Rothermel from Georgia Institute of Technology, Atlanta, USA. Every design type has number of already generated inputs that test C designs with different coverage. Error classes that can be found in Siemens Benchmarks are described in Table I.

During experiments FORENSIC tool's [1] Simulation-Based back-end was used, *simple* ranking and *one* fault assumption for Model-Based Error Localization were applied, with and without Dynamic Slicing. In table below measure of Error Localization accuracy is mean number of mutations required to correct errors in Siemens Benchmarks [10] during Mutation-Based Error Correction.

Table 4

Error Localization with and Without Dynamic Slicing for Siemens Benchmarks

Design	Percentage of corrected Designs	Mean Number of Correction mutations required to correct error when Error Localization is		Change, $\left\|\frac{A-B}{B}\right\| \cdot 100\,\%$
		with Dynamic Slicing (A)	without Dynamic Slicing (B)	

tcas	65,9	131,19	303,41	56,76
schedule	33,3	325,67	403,67	19,32
schedule2	33,3	460,67	495	6,93
replace	37,5	855,17	874	2,15
tot_info	46,9	781,33	793,47	1,52
print_tokens	14,2	825	891	7,40
print_tokens2	60,0	1255,17	1291,33	2,80

Designs that include Error Classes which are not supported by FORENSIC tool [1] were not included into experimental results.

Design internal structure defines how much Dynamic slicing reduces number of correction mutation required for Correction. If design has number of initializations in the beginning of the code, then *Dynamic Slicing* discards initializations that are not required for particular simulation with input *s*.

Accourding to experiments, *Dynamic Slicing* has strong influence on *tcas* design from Siemens Benchmarks, on *schedule*, *schedule2* and on *print_tokens*, for tcas design *Dynamic Slicing* algorithm will reduce number of required mutations for correction by 56,76 % in mean, for schedule – by 19,32 %, for schedule2 by *6*,93 %, and for print_tokens by 7,40 %.

Conclusions

The article describes *Error Localization* algorithm 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 [18], 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. 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.

References

1. *FORENSIC tool*, Available at: http://www.informatik.unibremen.de/agra/eng/forensic.php (accessed 23 January 2012).
2. Kamkin A., *Simulation-based hardware verification with time-abstract models*, Design & Test Symposium (EWDTS), 2011 9th East-West, 2011, pp. 43–47.
3. Abreu R., Zoeteweij P., and Van Gemund A.J.C., *An Evaluation of Similarity Coefficients for Software Fault Localization*, Dependable Comp., PRDC '06. 12th Pacific Rim Intern. Symposium, 2006, pp. 39–46.
4. *GIMPLE plug-in for GCC compiler*, Available at: http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html, (accessed 23 January 2012).
5. Weiser M., *Program Slicing*, Soft. Engineering, IEEE Transactions, 1984, pp. 352–357.
6. Kasyanov V.N., *Distinguishing Hammocks in a Directed Graph*, Soviet Math. Doklady, 1975, Vol. 16, no. 5, pp. 448–450.
7. Könighofer R. and Bloem R., *Automated error localization and correction for imperative programs*, Formal Methods in Computer-Aided Design (FMCAD), 2011, pp 91–100.
8. Jadish S. Gangolly, *Lecture notes on Design & Analysis of Accounting Information Systems*, University at Albany, NY, 2000.
9. William K. Lam, *Hardware Design Verification. Simulation and Formal Method-Based Approaches*, 2005.
10. *Bug Aristotle Analysis System – Siemens Programs, HR Variants*, Available at: http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/, (accessed 23 January 2012).
11. Debroy V., and Wong W.E., *Using Mutation to Automatically Suggest Fixes for Faulty Programs*, Soft. Testing, Verification and Validation (ICST), 3rd International Conf., 2010, pp. 65–74.
12. Chen M.Y., Kiciman E., Fratkin E., Fox A., Brewer E., *Pinpoint: Problem determination in large, dynamic internet services*, Proc. Int. Conf. on Dependable Systems and Networks, 2002, pp. 595–604.
13. Reps T., Ball T., Das M. and Larus J., *The use of program profiling for software maintenance with applications to the year 2000 problem*, Proc. 6th European Soft. Engineering Conf. (ESEC/FSE 97), Vol. 1301, LNCS, 1997, pp. 432–449.
14. Jain K., and Dubes R.C., *Algorithms for clustering data*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988, pp. 528–550.
15. J. de Kleer, and Williams B.C., *Diagnosis with behavioral modes*, Proc. IJCAI-89, 1989, pp. 1324–1330.
16. Reiter R., *A theory of diagnosis from first principles*, Artificial Intelligence, 1987, no. 32, pp. 57–95.
17. Quek C. and Leitch R.R., *Using energy constraints to generate fault candidates in model based diagnosis*, Intelligent Systems Engineering, 1st Intern. Conf., 1992, pp. 159–164.
18. Xiaoping J., *An approach to animating Z specifications*, Computer Software and Applications Conference, 1995. COMPSAC 95. Proc., 19th Annual Intern., 1995, pp. 108–113.
19. Repinski U. and Raik J., *Simulation-Based Verification with Model-Based Diagnosis of C Designs*, East-West Design Test Symposium, 2012, pp. 42–45.

УДК 62

ВЕРИФИКАЦИЯ НА ОСНОВЕ СИМУЛЯЦИИ С НАХОЖДЕНИЕМ И ИСПРАВЛЕНИЕМ ОШИБОК ДЛЯ С-ДИЗАЙНОВ

Урмас Репинский, ассистент
(Таллиннский технологический университет, Эхитаяте тээ, 5, г. Таллинн, 19086, Эстония, urrimus@hotmail.com)

Процесс верификации программного обеспечения позволяет удостовериться, соответствует ли дизайн своей спецификации. Анимация спецификации дает возможность реализовать верификацию на основе симуляции. Для локализации и исправления ошибки в случае неудачной верификации необходим доступ к структуре отлаживаемого дизайна. Для этого дизайн должен быть разобран в подходящий вид, в модель. Это минимально необходимые требования к инструменту, который сможет автоматически находить и исправлять ошибки в отлаживаемом дизайне.

Для локализации и исправления ошибок в дизайне также требуется реализация алгоритма симуляции модели. В данной статье представлены различные алгоритмы симуляции. Обычно применяется симуляция модели напрямую, когда модель дизайна симулируется с целью получения выводов из вводов, но такой подход не оправдывает себя, так как в этом случае функциональность языка программирования дизайна должна быть практически полностью заново реализована для симуляции. Автор описывает альтернативный подход – симуляция модели дизайна с помощью языка программирования дизайна. Он более оправдан, не требует повторной реализации функциональности, уже имеющейся в языке программирования дизайна, а также позволяет с легкостью реализовать алгоритм динамической нарезки для локализации ошибок. Представлены результаты локализации ошибок с использованием алгоритма динамической нарезки и без него.

Ключевые слова: *верификация на основе симуляции, диагностика ошибок, исправление ошибок, автоматическое исправление ошибок, отладка.*

Дигитальная система (дизайн) – это черный ящик, имеющий входы и выходы и скрытую внутреннюю структуру. Чтобы иметь возможность отлаживать систему, она должна быть представлена как белый ящик, то есть иметь подходящий разобранный вид. Разобранное представление системы сохраняется в соответствующей структуре, так называемой модели дизайна, которая позволяет отслеживать процессы во время выполнения системы и вносить изменения в ее структуру. Эти свойства модели необходимы для автоматической отладки системы – верификации, локализации и исправления ошибок. При локализации ошибок наблюдается след выполнения программы системы. Чтобы определить место в системе, ответственное за ошибку, требуется полный доступ к любой части ее модели для внесения соответствующих исправлений. Возможность устанавливать точки наблюдения в исследуемый дизайн позволяет тестировать любую часть дизайна, что необходимо для локализации ошибок. Это минимальные обязательные требования к инструменту, который сможет автоматически находить и исправлять ошибки в отлаживаемом дизайне [1].

С другой стороны, модель может рассматриваться как *интегрированная среда разработки* (ИСР) для реализации алгоритмов нахождения и исправления ошибок, то есть любая функциональность, которая может быть реализована, полностью зависит от свойств модели. Для отладки дизайнов, написанных на языке С, разработан инструмент FORENSIC (FOrmal Repair ENgine for SImple C), реализованный в рамках проекта FP7 DIAMOND [2] и удовлетворяющий всем требованиям к инструменту, перечисленным выше. FORENSIC предназначен не только для отладки, но и для разработки ПО и находится в открытом доступе в сети Интернет [3].

В составе инструмента автором реализованы алгоритм нахождения ошибок, выполняющий диагностику на основе модели, и алгоритм исправления ошибок, осуществляющий исправление на основе мутаций. Эти алгоритмы позволяют отлаживать довольно часто встречающиеся в реальных дизайнах простые ошибки, такие как неправильная арифметическая операция или ошибка в одном знаке в цифре. Алгоритм динамической нарезки позволяет повысить точность диагностики.

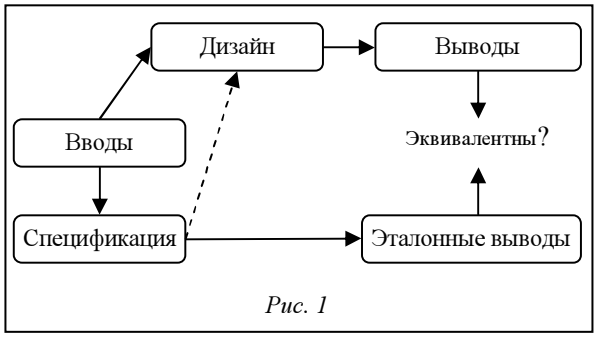

Рис. 1

Не менее важна проблема правильного представления исследуемого дизайна для верификации на основе симуляции. Одно из решений – использование формальных спецификаций, другое, более практичное – использование С-тестовых случаев [4]. В данной работе рассмотрены С-тестовые случаи, или С-исходный код без ошибки.

Реализованные в рамках проекта DIAMOND алгоритмы позволяют считать FORENSIC инструментом, применимым как для научных исследований, так и для создания реальных приложений и автоматической отладки ошибок, что позволит сократить время разработки и, следовательно, затраты на нее.

Верификация дизайна

По статистике, около 70 % времени, отведен-

ного на разработку проекта, занимает верификация дизайнов.

Существует две ее парадигмы – верификация на основе симуляции и формальная верификация. Основное их отличие в том, что для первой необходимы тестовые векторы, а для второй нет.

Парадигма верификации на основе симуляции включает четыре компонента: дизайн, тестовые векторы, эталонные выводы и механизм сравнения. Дизайн симулируется с тестовыми векторами, и результат сравнивается с эталонными выводами. Основной принцип верификации на основе симуляции показан на рисунке 1.

Формальная верификация разделяется на две категории – проверка эквивалентности (определяется, являются ли две реализации функционально эквивалентными) и верификация свойства (определяется, всегда ли имеется определенное свойство у реализации) [4]. Во время формальной верификации решение о том, являются ли реализации неэквивалентными, однозначно в отличие от верификации на основе симуляции, при которой результат зависит от выбора тестовых векторов для симуляции.

В данной работе описано использование верификации на основе симуляции.

Из парадигмы верификации на основе симуляции следует, что спецификация должна иметь формат, позволяющий проводить анимацию и получать эталонные выводы. Спецификация должна описывать дизайн на языке С и быть легко реализируемой.

Требования к спецификации

Как следует из рисунка 1, для проведения верификации на основе симуляции необходимы эталонные выводы, которые могут быть получены при анимации спецификации, оформленной в надлежащем формате.

В общем, спецификация – это документ, однозначно определяющий технические требования к предметам, материалам или сервисам. Спецификация в системном проектировании и в разработке программ – это описание системы, которая будет создана, руководство, необходимое тестерам для верификации. Кроме того, спецификация используется для верификации на основе симуляции.

Спецификация может быть формальной или неформальной. В процессе разработки информационных систем для бизнеса неформальные спецификации, использующие графическое моделирование, применялись со второй половины 1970-х годов. В последнее время были разработаны языки формальных спецификаций (Larch, VDM, Z, FOOPS и OBJ). Сегодня использование спецификаций находится на начальной стадии, но будет играть значительную роль в ближайшем будущем. Такие спецификации – это попытка математически описать структуру, функции и поведение информационных систем [5].

В случае неформальных спецификаций присутствует так называемый человеческий фактор, поскольку инженер верификации самостоятельно создает тестовые сценарии и гарантирует, что дизайн соответствует своей спецификации. С другой стороны, более целесообразно, а иногда и экономически выгоднее использовать формальные спецификации. При этом инженер верификации может следовать спецификации без написания дополнительного тестового кода или даже автоматически сравнивать эталонные выводы с выводами, полученными в процессе симуляции дизайна. Задача инженера во втором случае более ясна и может быть автоматизирована с помощью инструментов для автоматической верификации, и FORENSIC – один из них. Устранение влияния человеческого фактора на процесс верификации дизайнов может играть решающую роль в индустрии разработки ПО.

Спецификации можно разделить на две категории – определенные и неопределенные. Спецификации, которые можно анимировать, выводы которых зависят только от входов и никакие дополнительные допущения не используются, являются определенными [6].

В инструменте FORENSIC реализована верификация с исправлением ошибок на основе симуляции С-дизайнов и использован наиболее подходящий, по мнению автора, формат спецификации для такой системы – С-тестовые случаи или С-исходный код без ошибки. Для подобной спецификации дополнительная реализация анимации может быть заменена такой же симуляцией, какая применяется для верификации, что облегчает реализацию инструмента, а также позволяет задействовать одних и тех же специалистов для создания дизайна и спецификации, так как кроме знания языка программирования, никакие дополнительные знания не требуются. Спецификации, созданные на языках программирования высокого уровня (Java, C, C++), уже используются в индустрии ПО [4]. Язык, на котором реализована спецификация, называется языком верификации.

Реализации и функциональные компоненты FORENSIC

FORENSIC – реализация алгоритмов верификации на основе симуляции с локализацией и исправлением ошибок. Две различные реализации алгоритмов представлены на рисунке 2: на 2а показана основная реализация алгоритмов инструмента FORENSIC, а на 2б – более экономная. Отличие их в том, что в первом случае реализована симуляция отлаживаемого дизайна с целью получения выводов, а во втором такой независимой симуляции нет, реализация более экономная, по-

тому что и для получения выводов, и для диагностики, и для исправления ошибок требуется симуляция дизайна, и эти симуляции можно совместить. С другой стороны, реализация на рисунке 2а использует симуляцию исходного С-дизайна, а не симуляцию модели, что упрощает структуру инструмента. Но если дизайн разобран верно, без ошибок, выводы симуляции модели и дизайна должны быть идентичными.

Компоненты реализаций FORENSIC разбиты на несколько групп, которые отвечают за различную функциональность.

- Вводы-выводы – данные, необходимые для работы инструмента. На рисунке 2 контур блока – сплошная тонкая линия.
- Структуры данных, где сохраняется информация во время выполнения инструмента. Основная структура данных – модель инструмента FORENSIC.
- Функциональность, такая как внешний интерфейс, внутренний интерфейс, алгоритмы симуляции и анимации, алгоритмы локализации и исправления ошибок. Функциональность может быть дополнительно изменена или расширена, проект FORENSIC и исходные коды находятся в открытом доступе в сети Интернет.

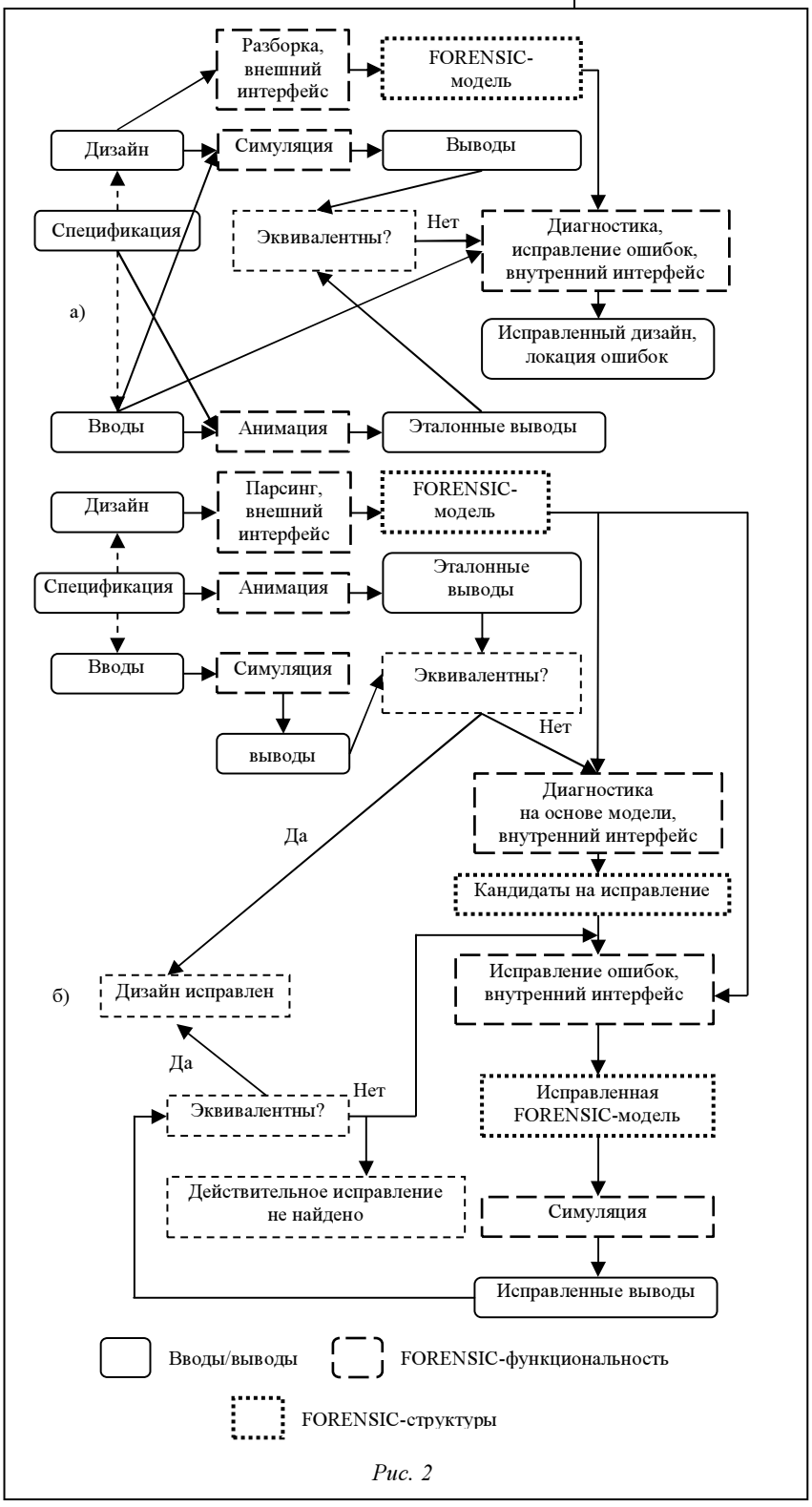

Рис. 2

Определение модели. Для верификации дизайна и последующей локализации (диагностики) и исправления ошибок модель, которая является подходящим описанием дизайна как белый ящик, должна быть разработана и реализована. В инструменте FORENSIC модель обрабатываемого дизайна – специальный случай направленного графа – гамак.

Определение 1. Диаграф – это структура $<N, E>$, где N – множество вершин, E – множество ребер, принадлежащих $N \times N$. Если (n, m) принадлежит E, то n – непосредственный предок m и m – непосредственный преемник n. Путь из вершины n к вершине m является списком длиной k таких вершин $p_0, p_1, ..., p_k$, что $p_0=n_1$, $p_k=n_2$, и для всех i, $1 \le i \le k-1$, (p_i, p_{i+1}) принадлежит E.

Определение 2. Направленный граф – это структура $<N, E, n_0>$, где $<N, E>$ – диаграф, n_0 принадлежит N и существует путь из n_0 ко всем остальным вершинам, принадлежащим N; n_0 называется начальной вершиной.

Определение 3. Гамак-граф – это структура $H=<N, E, n_0, n_e>$, которая удовлетворяет свойству, что $<N, E, n_0>$ и $<N, E^{-1}, n_e>$ являются направленными графами. Обра-

тим внимание, что, как обычно, $E^{-1}=\{(a, b)|(b, a)$ принадлежат $E\}$; n_e – конечная вершина [7].

Гамак является частным случаем направленного графа. Основное различие между ними в том, что гамак имеет одну конечную вершину, а направленный граф не обязан подчиняться этому требованию.

Алгоритм симуляции. Диагностика или локализация ошибок начинаются, если дизайн не соответствует своей спецификации, если выводы дизайна не согласуются с эталонными выводами и верификация (процесс, определяющий соответствие дизайна своей спецификации) требует симуляции дизайна. Диагностика сама по себе тоже требует симуляции модели как наиболее важного и трудоемкого компонента любого инструмента, который имел бы возможность находить ошибки в дизайне. Поэтому детально опишем реализованную в FORENSIC методологию симуляции.

Симуляция модели, являющейся гамаком, реализована в инструменте с использованием С-функциональности. Преимущество использования функциональности языка С для симуляции модели С-дизайна над симуляцией С-модели заключается в следующем.

Если использовать прямую симуляцию модели для получения выходных значений, то все логические операторы модели, присутствующие в дизайне, должны быть реализованы для симуляции и все значения переменных во время симуляции должны где-то сохраняться. Типы переменных могут быть различными – ссылки, массивы, сложные структуры данных и т.д. – практически модель хранения данных языка программирования С должна быть re-реализована, что трудоемко и проблематично.

Если же проводить симуляцию, используя С-функциональность, модель можно записать в исполняемый файл, инструментировать ее необходимыми операторами, а результат симуляции записать в файл, который будет доступен после симуляции. Это не требует re-реализации С-функциональности и позволяет получать любые необходимые данные о симуляции модели. При этом не теряется какая бы то ни было функциональность, но в результате имеется более быстрая симуляция, что критично, так как симуляция – наиболее важный компонент диагностики на базе модели и верификации на основе симуляции в общем. Кроме того, данная методология позволила реализовать механизм динамической нарезки для диагностики.

Такие же принципы можно применить и для симуляции Java- или VHDL-дизайнов.

Диагностика на основе модели. Определение 4. Активированные вершины N_a, принадлежащие N во время симуляции с вводами $i \in I$ графа гамак $H=<N, E, n_0, n_e>$ – это вершины, которые принадлежат активированному пути P из вершины n_0 к вершине n_e. Активированный путь P из вершины n к вершине m – список k – $p_0, p_1, ..., p_k$, такой, что $p_0=n_1$, $p_k=n_2$, и для всех i, $1 \le i \le k-1$, (p_i, p_{i+1}) принадлежит E.

Алгоритм 1. *Диагностика на основе модели.* Модель верифицируется на основе симуляции – симулируется с вводами I и для каждого ввода $i \in I$ выводы симуляции сравниваются с выводами эталонных выводов анимации спецификации. Если верификация не удалась, то есть выводы не соответствуют эталонным, для всех соответственных активированных вершин симуляции N_a увеличивается failed-счетчик, иначе увеличивается passed-счетчик. После симуляции алгоритмы ранжирования применяются к счетчикам и активированные вершины сортируются в соответствии с рангом. Выявленные в результате диагностики вершины с высоким рангом – это кандидаты на исправление.

Алгоритм динамической нарезки – техника, которая применяется после каждой симуляции с вводом $i \in I$ и позволяет удалить из активированного пути компоненты, не влияющие на результат симуляции. Эти компоненты могут быть константой или кодом дизайна и т.д.

Определение 5. Слой динамической нарезки включает в себя активированные вершины $N_s \in N$ гамак-графа $H=<N, E, n_0, n_e>$, имеющие влияние на результат симуляции. Вершины, находящиеся не в слое $\overline{N_s}=N-N_s$, могут быть удалены из активированного пути для симуляции с вводом i без изменения результата симуляции.

Для выявления таких вершин потребуются определения изменяемых и используемых переменных в компоненте – вершине гамак-графа.

Определение 6. Пусть V – множество переменных, существующих в дизайне P и в модели H дизайна P, которая является гамак-графом $H=<N, E, n_0, n_e>$. Тогда для каждой вершины $n \in N$ имеются два множества, каждое из которых принадлежит V: REF(n) – множество переменных со значениями, используемыми в n, DEF(n) – множество переменных со значениями, изменяемыми в n [8].

Алгоритм 1. *Динамическая нарезка.* Во время исполнения данного алгоритма вычисляются используемые глобальные переменные gREF на момент обработки каждого активированного компонента модели или активированной вершины $n \in N_a \in N$ гамак-графа $H=<N, E, n_0, n_e>$. Выполнение алгоритма осуществляется от конечной активированной вершины n_e к начальной n_0. На начало симуляции gREF=$\varnothing$.

Если компонент n – это С-функция или условие, то используемые переменные REF(n) в компоненте добавляются в множество глобальных используемых переменных gREF=gREF$\vee$REF(n). Точки наблюдения могут быть последними исполняемыми компонентами в модели, и переменные, которые выводятся точками, записываются в множество глобальных используемых переменных

gREF, так как точки наблюдения assert(...) или FORENSIC_...() – функции, что верно.

Если компонент n – присвоение, то используемые переменные REF(n) в компоненте добавляются в множество глобальных используемых переменных тогда и только тогда, когда изменяемые переменные DEF(n) принадлежат множеству глобальных используемых переменных gREF, иначе компонент находится не в слое, не имеет влияния на результат симуляции и может быть удален из списка активированных вершин. Определяемые переменные DEF(n) удаляются из списка глобальных используемых переменных, так как они переопределяются: DEF(n)$\in$gREF & gREF=(gREF – DEF(n)) $\vee$REF(n) $\wedge$ (gREF–DEF(n)), и n не в слое.

Пример реализации алгоритма динамической нарезки показан на рисунке 3.

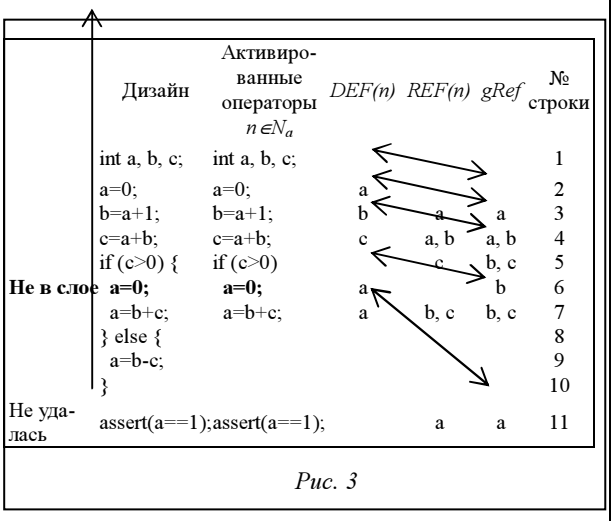

Рис. 3

Важно заметить, что переменные в дизайне могут быть различных типов, например, составными или частью составной переменной, поэтому использование имен для сохранения переменных в множествах REF(n), DEF(n) или gREF(n) непрактично. Гораздо удобнее сохранять уникальный адрес переменной, который может быть получен, если симуляция модели для диагностики и, следовательно, для динамической нарезки производится с использованием С-функциональности. Именно использование симуляции с использованием С-функциональности позволило практически реализовать алгоритм динамической нарезки.

Исправление ошибок. После окончания локализации можно попытаться исправить ошибку(и) в дизайне. Во внутреннем интерфейсе на основе симуляции FORENSIC реализован алгоритм исправления ошибок на основе мутаций.

Ошибки в дизайне делятся на множество классов, и различные методы исправления ошибок исправляют разные ошибки. Примеры ошибок в дизайне приведены в таблице 1.

Алгоритм исправления ошибок на основе мутаций заключается в следующем: компоненты – вершины гамак-графа $H=<N, E, n_0, n_e>$ мутируются и далее исправленная модель верифицируется на основе симуляции для проверки факта исправления мутацией ошибки. Если симуляция удалась, ошибка найдена и исправлена, если нет, мутируется очередной компонент или применяется следующий тип мутаций.

Мутации, реализованные в алгоритме исправления ошибок на основе мутаций back-end на базе симуляции FORENSIC, следующие.

- Мутации в операторах – операторы в кандидатах на исправление мутируются с операторами из той же группы операторов.
- Основные мутации констант. Это добавление 1 к константе (C→C+1), удаление 1 от константы (C→C-1), установление константы на ноль

Таблица 1

Класс ошибки	**Примеры**	**Поддержка back-end на основе симуляции FORENSIC**
AOR (arithmetic operator replacement)	ADD (+), SUB(-), MULT(*), DIV(/), MOD(%)	Да
ROR (relational operator replacement)	EQ(==), NEQ(!=), GT(>), LT(<), GE(≥), LE(≤)	Да
LCR (logical connector replacement)	AND (&&), OR(‖)	Да
ASOR (assignment operator replacement)	(+=), (-=), (=)	Да
Мутация целого числа	500→550, 740→700, 1→2, a[0]→a[1], n=0→n=1	Частично
Мутация числа с плавающей запятой	0,0→0,1, 0,0→1,0	Частично
Неверный индекс массива	$a[i$-1]→a[i], $a[i$+1]→$a[i]$, a[3]→a[4], $a[var]$→a[0]	Частично
+1 добавлен/отсутствует	r=i→r=i+1, $f(a, b, c$+1)→$f(a, b, c)$, $if(i)$→$if(i$+1)	Частично
Мутация константы (константа заменена другой константой)	TRUE→FALSE	Нет
Мутация числа с плавающей запятой	2,5066282→2,50663 EPS→EPS-0,000001	Частично
Код отсутствует/Код добавлен	/* && (Cur_Vertical_Sep>MAXALTDIFF); missing code */	Нет
Функция заменена константой	Inhibit_Biased_Climb()→Up_Separation	Нет
Изменение логики	Brackets removed	Частично
Ошибка в операторах ввода-вывода	fprinf, getch	Да

(C→0) и отрицание константы (C→-C).

- Мутации в цифрах констант. Значения 0, ..., 9 заменяют каждую цифру десятичного значения константы по очереди. Мутация применяется ко всем константам независимо от того, является ли ее значение целым числом или с плавающей запятой.
- Округление цифр с плавающей запятой. Цифры с плавающей запятой (в C++ float и тип double) заменяются округленными (округление применяется вверх и вниз).

Для мутаций в операторах операторы поделены на следующие группы.

- Арифметические операторы: плюс (+), минус (-), умножение (*), деление (/), модуль (%).
- Операторы присваивания: присваивание (=), присваивание плюс (+=), присваивание минус (-=), присваивание умножение (*=), присваивание деление (/=), присваивание модуль (%=).
- Операторы сравнения: меньше (<), больше (>), равно (==), не равно (!=), меньше равно (≤), больше равно (≥).
- Логические операторы: логическое И (&&), логическое ИЛИ (||).
- Операторы смещения битов: смещение битов налево (<<), смещение битов направо (>>), битовое И (&), битовое ИЛИ (|), битовое ИЛИ-исключающее (^).
- Операторы смещения битов с присваиванием: смещение битов налево с присваиванием (<<=), смещение битов направо с присваиванием (>>=), битовое И с присваиванием (&=), битовое ИЛИ с присваиванием (|=), битовое ИЛИ-исключающее с присваиванием (^=).
- Унарные арифметические операторы: унарный плюс (+), унарный минус (-).
- Операторы повышения/понижения: повышение до (++*x*), повышение после (x++), понижение до (--*x*), понижение после (*x*--).
- Унарные логические операторы: логическое

Таблица 3

Дизайн с ошибкой	Количество вводов, которые выявили ошибку	Количество мутаций, необходимое для исправления			Описание ошибки
		Диагностика с нарезкой	Диагностика без нарезки	Исправление без диагностики	
tcas/v1	132	325	524	686	ROR
tcas/v2	69	65	259	264	Мутация в константе
tcas/v3	23	322	521	1035	LCR
tcas/v4	26	244	443	861	LCR
tcas/v5	10	Не найдено			Отсутствует код
tcas/v6	12	43	242	292	ROR
tcas/v7	36	36	83	83	Мутация в целом числе
tcas/v8	1	35	180	180	Мутация в целом числе
tcas/v9	9	83	282	332	ROR
tcas/v10	14	43	242	292	ROR
tcas/v11	14	44	243	293	ROR
tcas/v12	70	163	362	935	LCR
tcas/v13	4	195	394	883	Мутация в целом числе
tcas/v14	50	90	284	911	Мутация в целом числе
tcas/v15	10	Не найдено			Отсутствует код
tcas/v16	70	18	18	18	Мутация в целом числе
tcas/v17	35	20	67	67	Мутация в целом числе
tcas/v18	29	36	132	132	Мутация в целом числе
tcas/v19	19	39	184	184	Мутация в целом числе
tcas/v20	18	168	367	585	ROR
tcas/v21	16	Не найдено			Функция заменена константой
tcas/v22	11	Не найдено			Функция заменена константой
tcas/v23	42	Не найдено			Функция заменена константой
tcas/v24	11	Не найдено			Функция заменена константой
tcas/v25	3	168	367	551	ROR
tcas/v26	11	Не найдено			Отсутствует код
tcas/v27	10	Не найдено			Отсутствует код
tcas/v28	76	6	200	205	Изменение логики
tcas/v29	18	Не найдено			Функция заменена константой
tcas/v30	58	Не найдено			Функция заменена константой
tcas/v31	14	174	368	613	Добавлен код
tcas/v32	2	45	239	314	Добавлен код
tcas/v33	89	Не найдено			Мутация в целом числе
tcas/v34	77	Не найдено			Изменение логики
tcas/v35	76	6	200	205	Изменение логики
tcas/v36	124	706	926	1343	Мутация в целом числе
tcas/v37	93	Не найдено			Неверный индекс массива
tcas/v38	91	Не найдено			Мутация в целом числе
tcas/v39	3	168	367	551	ROR
tcas/v40	126	211	410	687	Изменение логики
tcas/v41	26	89	288	434	Изменение логики
Максимум		706	926	1343	Всего исправлено 27/41
Среднее		127,89	292,74	463,04	
Минимум		20	67	67	
Сумма		3542	8192	12936	

НЕТ (!), битовое дополнение (~).

На основе мутаций невозможно исправить сложные ошибки, но этот алгоритм имеет свои преимущества. Классы ошибок, поддерживаемые алгоритмом исправления ошибок на основе мутаций, часто встречаются в реальных программах, кроме того, алгоритм относительно быстрый и надежный.

Экспериментальные данные

Эксперименты для проверки работоспособности системы были проведены с использованием дизайнов, разработанных Исследовательским центром Сименс и доработанных Мэри Жан Харрольд и Грегг Ротермель [9]. Данные дизайны написаны на языке С и, следовательно, подходят для исправления с помощью FORENSIC.

Описание дизайнов. Дизайны – это С-программы с разными функциональностями:

tcas – логические/арифметические вычисления;

print_tokens, print_tokens2 – обработка текста;

replace – обработка текста;

schedule, schedule2 – использование С-структур и объединений;

tot_info – арифметические вычисления.

Более детальное описание дизайнов приведено в таблице 2.

Таблица 2

Дизайн	Количество строк	Количество вводов	Количество дизайнов с ошибками
tcas	186	1605	41
schedule	419	2650	9
schedule2	312	2710	9
replace	558	5543	31
tot_info	413	1052	22
print_tokens	569	4130	7
print_tokens2	515	4115	10

Результаты экспериментов исправлений на основе мутаций для tcas-дизайна представлены в таблице 3.

Для данного дизайна алгоритм нарезки позволил снизить количество мутаций для исправления ошибок в среднем на 55,5 %. Это обусловлено тем, что дизайн имеет инициализации в начале кода, из которых только часть необходима для обнаружения ошибок в различных дизайнах. Процесс динамической нарезки позволяет удалить эти инициализации из списка кандидатов для исправления, что значительно повышает скорость исправления, уменьшая количество необходимых мутаций.

Эксперименты для остальных дизайнов аналогичны.

Сводка экспериментальных результатов отражена в таблице 4.

Таблица 4

Дизайн	Количество исправленных дизайнов по отношению к общему количеству дизайнов, %
tcas	27/41=65,85
schedule	3/9=33,33
schedule2	3/10=30,00
replace	12/32=37,50
tot_info	15/32=46,88
print_tokens	1/7=14,29
print_tokens2	7/10=70,00

Сравнение экспериментальных результатов

В работе [10] представлены эксперименты исправлений на основе мутаций с использованием дизайнов [9].

В таблице 5 результаты из [10] сравниваются с экспериментальными результатами с использованием FORENSIC.

Таблица 5

Дизайн	A	B	C	D
tcas	9	27	41	43,9
schedule	0	3	9	33,33
schedule2	1	3	9	22,22
replace	3	12	32	28,13
tot_info	8	15	23	30,43
print_tokens	0	1	7	14,29
print_tokens2	0	7	10	70,00
Общее количество	21	68	131	35,88

Примечание: A – количество дизайнов, исправленных в [10]; B – количество дизайнов, исправленных с помощью FORENSIC back-end на основе симуляции; C – всего дизайнов; D – количество дизайнов, исправленных с использованием FORENSIC дополнительно к [10], (B-A)*100/C, %.

Из таблицы можно заключить, что FORENSIC back-end на основе симуляции в среднем исправляет на 35,88 % ошибок больше.

Итак, необходимо отметить, что процесс разработки ПО сложен и затратен. Любой инструмент, упрощающий этот процесс и уменьшающий затраты, полезен.

FORENSIC – попытка разработки такого инструмента. На момент релизации в январе 2012 г. верификация и исправление ошибок поддерживались только для С-дизайнов, алгоритмы нахождения и исправления еще не стандартизированы, не определен формат спецификаций, но любые аспекты программы при необходимости могут быть модернизированы. В проект несложно включить также поддержку C/C++ и SystemC-дизайнов. Когда использование спецификаций будет автоматизировано, подобные инструменты станут предметами первой необходимости в процессе разработки программ.

В заключение следует перечислить основные свойства инструмента FORENSIC.

1. Выбор модели представления кода – важный аспект инструмента, структура представления

влияет на функциональность.

2. Формат спецификаций отлаживаемого дизайна должен поддерживать анимацию.

3. Алгоритм диагностики может быть простым, как диагностика на основе мутаций. Различные алгоритмы диагностики могут применяться, если в дизайне не одна, а несколько ошибок.

4. Алгоритм исправления ошибок должен поддерживать исправление максимального количества типов ошибок, и следовательно, алгоритм исправления ошибок может быть сложным, то есть состоять из нескольких алгоритмов, примененных один за другим.

5. Симуляция и анимация – это функциональности, которые занимают наибольшее количество времени. Использование С-функциональности для симуляции и анимации является наиболее разумным подходом, позволяющим получать максимальную надежность и скорость. Кроме того, С-функциональность для симуляции дает возможность реализовать динамическую нарезку для диагностики.

Литература

1. Chepurov A., Jenihhin M., Raik J., Repinski U., Ubar R. High-level design error diagnosis using backtrace on decision diagrams. 28th Norchip Conf., Tampere, FINLAND, 2010.

2. Diamond FP7 Project. URL: http://fp7-diamond.eu/ (дата обращения: 4.05.2012).

3. University of Bremen, Faculty of Computer Science. URL: http://www.informatik.uni-bremen.de/agra/eng/forensic.php (дата обращения: 4.05.2012).

4. William K. Lam: Hardware Design Verification. Simulation and Formal Method- Based Approaches. Prentice Hall. Profession Technical Reference, 2005, pp. 1–23.

5. Jadish S. Lecture notes on Design & Analysys of Accountuing Information Systems, URL: http://www.albany.edu/acc/courses/acc681.fall00/681book/681book.html (дата обращения: 02.03.2012).

6. An approach to animating Z specifications, Xiaoping Jia, Computer Software and Applications Conf., COMPSAC-95. Proceedings., Nineteenth Annual International, 1995, pp. 108–113.

7. Kas'janov V.N. Distinguishing Hammocks in a Directed Graph, Soviet Math. Doklady, 1975, Vol. 16, no. 5, pp. 448–450.

8. Mark Weiser, Program Slicing, IEEE Transactions on Software Engineering, 1984, Vol. SE-10, no. 4, pp. 352–357.

9. Bug Aristotle Analysis System – Siemens Programs, HR Variant, URL: http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/ (дата обращения: 02.03.2012).

10. Debroy V. and Wong W.E. Using Mutation to Automatically Suggest Fixes for Faulty Programs, Software Testing, Verification and Validation (ICST), 3rd International Conf., 2010, pp. 65–74.

11. Kang J., Seth S.C., Yi-Shing Chang, Gangaram V. Efficient Selection of Observation Points for Functional Tests. Quality Electronic Design, ISQED-2008, 9th International Symposium on, 2008, pp. 236–241.

УДК 519.688

АЛГОРИТМ ПЛОТНОЙ СТЕРЕОРЕКОНСТРУКЦИИ НА ОСНОВЕ КОНТРОЛЬНЫХ ТОЧЕК И РАЗМЕТКИ ПЛОСКОСТЯМИ

Г.Р. Кривовязь, аспирант; С.В. Птенцов, студент; А.С. Конушин, к.ф.-м.н., научный сотрудник

(Московский государственный университет им. М.В. Ломоносова, Ленинские горы, г. Москва, 119991, Россия,

gkrivovyaz@graphics.cs.msu.ru, sptentsov@rambler.ru, ktosh@graphics.cs.msu.ru)

Предлагается новый алгоритм плотной стереореконструкции по паре изображений. Алгоритм отталкивается от предложенной в одной из современных работ идеи агрегации стоимостей в каждой точке не по локальной окрестности, а по минимальному покрывающему дереву, охватывающему все пиксели изображения. При этом ключевой особенностью предложенного алгоритма является переход из пространства диспаритетов в пространство плоскостей: решение в каждом пикселе выводится из уравнения присвоенной ему плоскости. В статье также предлагается способ вычисления набора плоскостей, которыми будет аппроксимироваться трехмерная сцена, на основе контрольных точек. Контрольные точки, получаемые путем сопоставления изображений, используются и для регуляризации решения. Приводятся результаты тестирования предложенного алгоритма на современном, опубликованном в 2012 году тестовом наборе, содержащем 194 пары изображений наземной городской съемки. Кроме того, обсуждаются возможные пути дальнейшего развития метода.

Ключевые слова: *компьютерное зрение, плотное стерео, карта диспаритета, стереопара, контрольные точки, сопоставление изображений.*

A DENSE STEREO MATCHING ALGORITHM BASED ON GROUND CONTROL POINTS AND PLANE LABELING

Krivovyaz G.R.*, Postgraduate;* ***Ptentsov S.V.****, Student;* ***Konushin A.S.****, Ph.D., Research associate*

(Lomonosov Moscow State University, Leninskie Gory, Moscow, 119991, Russia,

gkrivovyaz@graphics.cs.msu.ru, sptentsov@rambler.ru, ktosh@graphics.cs.msu.ru)

Abstract. In this work a new dense stereo matching algorithm is proposed. The algorithm is based upon a recently introduced idea of non-local cost aggregation on a minimum spanning tree that includes all image pixels.