Generic Error Localization for the Electronic System Level

Sebastian Pointner* Pablo González de Aledo† Robert Wille*
*Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
†Imperial College London, UK
Email: sebastian.pointner@jku.at pablo.aledo@gmail.com robert.wille@jku.at

***Abstract*—Several methods and tools have been proposed which supports designers in verifying embedded systems in early phases of the design process, e.g. at the *Electronic System Level* (ESL). However, they only show whether an error indeed exists in the system, but it frequently remains open to efficiently locate the source of this error. In this work, we propose a generic error localization methodology. More precisely, by applying code augmentations and conducting further runs of the verification method, it is analyzed what statements may have caused the error. The respectively determined statements then pin-point the verification engineer to possible error locations. By conducing all this on the code level only, the proposed methodology can be applied to any verification method available today. The suitability of the proposed methodology is demonstrated by means of a verification flow based on symbolic execution.**

I. Introduction

Nowadays, verification is an essential step within the design of embedded systems and should already begin in early phases of the design process, e.g. at the *Electronic System Level* (ESL, [1]). To this end, engineers are applying sophisticated methodologies and tools to ensure the correct functionality of a new system, e.g. assertion checkers [2], [3], property checkers [4] or symbolic executers [5], [6], [7]. However, once it has been identified that a considered system includes an error (e.g. because an assertion could not be satisfied or a verification method generated a counterexample violating a property), it frequently remains open to efficiently locate the source of this error. Since modern embedded systems are getting steadily more complex, localizing errors has become a very complex task. In many cases, huge parts of the source code have to be inspected for this – usually a manually conducted and, hence, rather time-consuming task. In fact, checking the correctness of a system and identifying the location of errors nowadays consumes significantly more time than the actual design process itself [8].

Motivated by that and in an attempt to reduce the overall verification and debugging time, some few methods have been proposed which aid designers in locating the source of an error. For example, in [9] the authors propose an automatic debugging environment which is build up on the GCC compiler front-end and a special graph-representation which eventually is used as starting point for symbolic execution. Compared to this approach, the authors of [10] propose an approach for error localization based on the Maximum Satisfiability (MAX-SAT) problem. Their idea is to use the control flow graph of any program as an initial point and to extract traces which eventually will be formulated in terms of an MAX-SAT instance. In [11], the authors propose an approach based on introducing a kind of oracle function which is used to replace the right hand site of assignment statements in order to find possible locations capable for violating the program execution while in [12] the authors try to identify errors in traces by invoking so called *correctness functions* which are used to classify the according traces in terms of transitions.

However, related work on error localization at the ESL in general is still significantly restricted to a particular application domain and/or requires a huge modification of the respectively utilized core methods such as symbolic simulation or reasoning engines. The latter particularly makes those solutions dependent on the current version of both, the code in which the verification approach is implemented as well as the current version of the core methods. Future releases/updates will likely render those solutions non-applicable or require substantial efforts for adjustment.

In this work, we propose the idea of a generic error localization methodology which overcomes these problems. The main idea is to impose strategies for error localization directly at the code level in which the considered system is implemented and, afterwards, utilize core methods only in a "black box" fashion (i.e. without interfering with their internals). To this end, we automatically augment an erroneous implementation of a system with so called *replacement*-variables which allows to replace the execution of a single statement with an arbitrary one. Then, arbitrary verification methods can be applied (e.g. assertion checkers, property checkers or symbolic execution methods which have been applied to identify the existence of the error in the first place). As they now are "allowed" to replace certain statements with arbitrary ones, they will generate execution traces satisfying the failing assertion or property. By analyzing what statements have been replaced, the designer gets pin-pointed to possible sources of the error. By conducing all this only on the code level, the proposed methodology can be applied to any verification method available today (including possible updates and newly proposed ones in the future). The suitability of the proposed methodology is demonstrated by means of a verification flow based on symbolic execution.

II. General Idea

In this section, we will illustrate the general idea of the proposed methodology for a generic error localization. To this end, we are considering the implementation of a small function from a system as running example. Note that this function as well as the considered error are kept simple in order to ease the corresponding descriptions. The proposed methodology can eventually be applied to all (practically relevant) functions/system implementations which can be handled by the respectively applied verification method (see also discussions in Section III and Section IV).

Example 1. *Consider the code snippet shown in Figure 1 which is supposed to provide the implementation of a function* abs() *calculating the absolute value of a given integer number. Here the user has made an error where instead of assign the inverted value, the negative value has been assigned. Applying a verification method such as an assertion checker, a property checker, a symbolic executer, or similar would e.g. yield the following counterexample: Applying* -1 *as input returns* -1 *and not* 1 *as expected. The goal is now how to efficiently locate the source of this error, i.e. statements which could indicate to the error.*

To determine possible statements which might explain the error (or are responsible for the error), we can now utilize the same verification method which has been applied to detect the error in the first place. To this end, we particularly consider the respectively obtained counterexample (i.e. the determined input/output which showed the existence of the error) and enforce the input of the counterexample as well as the actually desired output to the function. If we would now apply the verification method again (i.e. checking whether the erroneous implementation generates the correct output for an input which already has been determined as triggering the error), the verification method obviously would not find a possible execution trace.

```
1  int abs (int const input){
2    if(input < 0)
3    return input;
4    else
5    return input;
6  }
7
8  int main() {
9    int signed_val = -42;
10   int unsigned_val = abs(signed_val);
11
12   assert(unsigned_val >= 0);
13
14   return 0;
15 }
```

Fig. 1: Erroneous implementation of abs().

Example 2. *Consider again the code from Figure 1. Setting the input of the* abs() *function to* -1 *obviously is a counterexample, since the function would generate an output of* -1*, although an output of* 1 *is desired. Now conducting e.g. a symbolic execution (as one representative of an applied verification method) while, at the same time, enforcing input to* -1 *and the output to be a positive integer number, no executable trace can be generated.*

Next, we augment the code itself in order to allow for the generation of traces. To that end, for each $stmn_i$, we introduce an additional variable $r_i \in \mathbb{B}$ called *replacement variable* to alter the statement $stmns_i$ used in the program in terms of $\overline{r}_i \rightarrow stmnt_i$ (this is similar to approaches used e.g. in [13], [14]). If the verification method now sets $r_i = 0$, the statement $stmnt_i$ will be enforced as in the original implementation. If in contrast the verification method sets $r_i = 1$, an arbitrarily new value for $stmnt_i$ can be employed. This may eventually allow for the generation of an execution trace which, although a counterexample is applied, generates the desired (i.e. correct) value. In the latter case, the correspondingly modified statement $stmnt_i$ serves as possible source of the error or, at least, could help the designer to locate the source of the error.

However, the verification method now could alter the entire set of statements – by simply setting an arbitrary number of the replacement variables r_i to 1. In order to avoid that, we limit the number of replacement variables set to 1, i.e. we enforce $\sum_{r_i} = k$ (using so-called *cardinality constraints* [15]) where k can initially be set to 1 and accordingly increased in order to generate further traces.

Following this scheme, the verification method applied to detect the existence of an error can now also be applied to localize the source of it. In fact, if the verification method determines valid execution traces, these traces must include a statement whose corresponding r_i-variable has been set to 1. The activation of the r_i-variable for a certain statement now shows that altering this statement is capable to influence the functions behaviour in such a manner that the enforced counterexample nevertheless generates the desired output. Accordingly, this statement may pinpoint the designer to a possible error location.

Example 3. *Consider again the code shown in Figure. 1 and the counterexample used in Example 2. Introducing replacement variables* $r_1, r_2, r_3 \in \mathbb{B}$ *for the statements shown in lines 2,3 and 5 of Figure 1, respectively, and extending these statements with the implications as described above yields an extended version of the original program. Passing this program to the verification method yields an executable trace when* r_2 *is set to 1 (allowing for the assignment of an arbitrary value for variable* output *at this point which eventually allows to generate the desired output* 1*). Hence, line 3 may indicate a possible error.*

Applying this approach more than once, may lead to further $r_i = 1$ assignments and, hence, may pinpoint to further statements (and, hence, locations) in the code that may lead to the desired results. Moreover, we could increase the number of r_i variables set to 1 to generate further assignments (might be necessary, if e.g. multiple errors occur which prevent the generation of the desired result). Overall, this pinpoints the designer to several locations which are worth inspecting. Those locations can eventually be utilized because:

- they already pinpoint to the actual error,
- they, at least, reduce the number of possible statements to be considered for debugging since not the entire code has to be inspected but only the determined statements (because only changes here eventually allow for the desired result), and
- they provide starting points for further debugging methods such as slicing [16], [17].

III. Representative Realization

The methodology proposed above can, in principle, be realized on top of every available verification method that initially generated the respective counterexample showing the existence of the error. In this section, we discuss an according realization. To this end, we consider symbolic execution as a representative verification method. The following provides the corresponding details, i.e. we briefly review symbolic execution as a verification method, describe the implementation of the proposed error localization approach on top of the symbolic execution engine *Forest* [6], and, finally, discuss further possible optimizations for this particular realization.

A. Symbolic Execution

Symbolic execution [18], [19] is an advanced program analysis technique in which, instead of evaluating the execution of the program by applying the semantics of the language to precise values of the variables (i.e. explicit simulation), placeholders are used which can hold any value. Symbolic execution keeps track of all symbolic values in terms of symbols and constants for every variable. Each time the execution of a program reaches a branching condition, the program forks and symbolic execution considers both outcomes of the branching condition. For each existing execution path, those branch conditions will eventually be utilized to construct so-called *path conditions*, which symbolically represent the constraints that are associated to a value in a particular state of the program. Even when the main use of symbolic execution has been test-case generation for reachability [20], the uses of symbolic execution covers many domains such as worst-case execution time determination [21] or equivalence-testing [22].

Example 4. *To illustrate the application of symbolic execution, we apply the technique to the function introduced in Figure 1. Recall that the function is build up out of 4 lines of code containing three basic blocks and one branch. Due to this branch, there are two possible execution paths of this function:*

$$abs(input) = \begin{cases} input < 0 & return\ -input \\ else & return\ input \end{cases} \quad (1)$$

Symbolic execution engines like KLEE [5] or Forest [6] are able to systematically explore feasible paths in program functions by combining heuristic-based search and satisfiability solvers like [23], [24]. In the following, we will rely on the symbolic execution engine *Forest*, which implements symbolic execution for the LLVM *Intermediate Representation* (IR) [25].

For our work, we rely on the LLVM intermediate representation as a common representation to both introduce the error localization primitives as well as applying the verification technique. LLVM IR is a low-level language with precise semantics to which many high-level languages can be translated. Supporting LLVM IR as a common language for instrumentation and verification enables our technique to be applicable for many verification frameworks. Indeed, if a tool implements the verification of reachability conditions in the LLVM IR – due to the fact that the instrumentation for error location is decoupled from the verification itself –, it can also be used for error localization.

Example 5. *Consider again the running example introduced in Figure 1. In this function, we have used the function call* assert(). *This function call tells Forest that it should try to come up with a counterexample to satisfy and to violate the assertion. By invoking Forest together with the assertion from line 12 of Figure 1, we are able to directly generate a counterexample violating the assertion.*

B. Error Localization on Top of Forest

Having the counterexample, we now aim to determine possible error locations by realizing the methodology proposed in Section II on top of Forest. Since Forest is based on the LLVM IR, we first review the corresponding input and how to generate it. Afterwards, we introduce how the corresponding *replacement variables* (eventually employing $\overline{r}_i \rightarrow stmnt_i$ for each statement $stmnt_i$) are incorporated to alter statements and allowing the replacement of the value used by statement $stmnt_i$.

The basic idea of LLVM [25] is to introduce a virtual machine for system languages. LLVM is based on a three layer approach where (1) the top layer, the so-called *front end*, translates the system language program code into an (2) intermediate format which can then be further processed as well as optimized, and (3) eventually be translated into machine language by a *back-end* available for the needed target architecture. The LLVM IR representation can be compared to the assembly language. Moreover, by translating any C program into the LLVM IR level, most of the complex instructions have already been broken down into basic operations (e.g. load, branch, add). Since Forest is capable for using C code or the LLVM IR representation of any program as input, it deems suitable to introduce the proposed replacement-variables at this layer as well. To this end, the respectively given code has first to be translated to the LLVM IR representation which can be done by applying an LLVM front-end like Clang.

Next, the resulting LLVM code shall be augmented with the corresponding replacement variables. For processing the program in the LLVM IR layer, LLVM offers the application of so-called *optimization passes*. By applying an *optimization pass*, the program can be altered at the LLVM IR layer which makes the optimization completely independent of any programming language or target architecture. In oder to realize the introduction of the replacement-variables, we have implemented an *optimization pass* capable to handle the augmentation. Finally, Forest is capable to apply additional *optimization passes* out of the box. To this end, we could realize the augmentation based on the *optimization pass* and could directly hand the pass over to Forest which applies the *optimization pass* before it starts to prepare the code for the symbolic execution.

After Forest has completed the symbolic execution of the program which has been augmented with the replacement-variables, we can obtain the generated results. Forest stores all evaluated traces and, we can now inspect these traces for activated *replacement variables*. Again, if Forest activated a *replacement variable* in order to alter a trace, we can use this information as an indicator for a possible error within the statement which has been altered by the *replacement variable*.

Example 6. *Consider again the running example from above (i.e. the broken implementation of the abs() function as shown in Figure 1). By applying the proposed* optimization pass*, we are now introducing the replacement variables for each statement. Therefore our* optimization pass *is realizing the* $\overline{r}_i \rightarrow stmnt_i$ *transaction by employing an* if-then-else *selection operator. This finally allows us to replace the content of the statement by eventually activating the transaction.*

Forest is now going to perform symbolic execution on top of this augmented *LLVM IR version of the broken abs() function. In order to explore every possible trace through our* augmented *version of the abs() function, Forest is now constructing possible traces which then will be evaluated. If a trace can be evaluated to true only when the reasoning engine applied by Forest has to activate an* replacement variable*, this trace might possibly pinpoint to an error location.*

C. Statements to be Instrumented

In this section, now we finally discuss what statements should be considered to get augmented by a replacement-variable and its corresponding program modification.

To this end, recall that the corresponding augmentation with replacement-variables should be applied to statements which are capable of influencing the program's behavior in a fashion so that they are actually able to generate the desired outputs (although the counterexample is applied). In an ideal scenario, of course all statements should be considered for this (one of them definitely is responsible for the failed assertion output). However, that would lead to scalability issues. In fact, adding only one *replacement variable* ends up in doubling the search space to consider. Doing that for *every* statement will easily become infeasible. Therefore, a limitation of what statements to be augmented is needed in order to prevent an exponential number of traces (namely $2^{\text{number\_of\_selected\_statements}}$) to be generated.

To this end, we propose a selection scheme which limits the number of augmented statements. For this purpose, we are focusing on the program in terms of the *control flow graph* and the *data flow graph*. By focusing on those graph representations of the program, we could identify statements which are more likely of influencing the program execution (and, hence, are also more likely to violate any assertion). This is basically based on previous evaluations conducted in [26]. Here, the authors concluded that common flow patterns in terms of control and data flow are likely to cause the error of a program. In the following, we describe these influential statements in somewhat more detail.

Control Flow Operator: The *compare* operator has been identified as the most influencing operator of the control flow. It is capable to change the programs behavior by routing the execution through a different trace of the program than the desired one. Because of this, we are focusing our error localization approach to those *compare* operators.

Data Flow Operator: Compared to control flow, where we were able to classify one single operator as the most influencing operator, the data-flow is more versatile. Hence, instead of identifying one single operator we consider a class of most influencing operators. Those operators, namely *binary operators*, are mostly invoked into data flow manipulations. *Binary operators* can be represented by arithmetic operators like *addition* or logic operators like *exclusive or*.

Based on the discussions above, a compromise between a full consideration of all statements and a restricted consideration of statements is provided. In fact, by considering only the data flow and control flow operators as shown above, we focus on the most influencing operators. This significantly reduces the complexity of the error localization methodology while, as confirmed by evaluations summarized in the following section, still yields useful results that, indeed, pinpoint the designer to the actual error location.

IV. Demonstration and Application

In this section, we demonstrate the applicability of the proposed methodology. To this end, we have implemented the approach as described in the previous section as a representative. As test cases, we employed instances from the Siemens' *Traffic Collision Avoidance System* (TCAS) benchmark suite [27]. This benchmark suit offers different versions of an algorithm for a traffic collision detection system into which different errors have been injected. In the following, we demonstrate how the determination of the location of these errors can be improved with the proposed methodology. More precisely, we first discuss this from a user perspective for a single case. Afterwards, a summary of further cases which have been considered is provided.

TABLE I: TCAS Experiments.

Input		Result		
Benchmark	Type	Time [s]	Candidates	Match
v1	operator mutation	66	8	✓
v2	operator mutation	24	6	✓
v3	logic change	20	3	✓
v4	logic change	76	11	✓
v5	missing code	14	6	✓
v6	logic change	38	5	✓
v20	logic change	23	8	✓
v28	branch manipulation	365	13	✓
v39	operator mutation	570	13	✓
v41	wrong assignment	68	16	✓

A. User View

In order to illustrate the application of the proposed methodology from a users perspective, we have chosen one of the benchmarks from the TCAS suite, namely $v3$. Here, a logic change has been injected into the algorithm (i.e. a Boolean AND-operation has been replaced by a Boolean OR-operation). Applying the proposed methodology (using the Forest realization as introduced in Section III) eventually yields three possible error locations, i.e. statements which could help the designer in understanding the source of the errors. In a consequently following step, the designer could use this information now by e.g. setting breakpoints for those statements to investigate their influence even further. By this, he/she is eventually pin-pointed to the actual error. Although this is a rather small example, it illustrates how the designer can utilize the proposed methodology. In fact, he/she does not need to change the respectively applied verification method or to step through the single steps of a counterexample. Instead, the designer is directly pinpointed to possible error locations out of which he/she can easily derive the actual source of the error.

B. Further Cases

Besides the case from above, we additionally applied the proposed methodology to further benchmarks from the TCAS benchmark suite. Correspondingly obtained results from those case studies are summarized in Table I. Here, the first columns provide the name of the respectively considered benchmarks as well as the type of error which has been injected here. Afterwards, we list the runtime needed by the proposed methodology in order to determine the error candidates. the number of obtained candidates, and whether the actual reason of the error was within the set of obtained error candidates (i.e. whether the error candidates indeed matched the error). The results clearly show the benefit of the proposed methodology. For all cases, the proposed approach was capable of reducing the set of statements to consider to a rather small number which can easily be inspected by the designer. Moreover the actual error was always be amongst those determined statements. As for run-time performance, the considered realization (using the symbolic executer Forest as verification method) was able to determine the error candidates in reasonable time.

Overall, the proposed methodology was able to significantly support the designer in determining the error of the considered benchmarks.

V. Conclusion

The ever increasing complexity of modern systems remains a significant challenge. Therefore, verification engineers are trying to perform the verification of new systems already in early design phases. However, every time when an error has been unveiled by a certain verification method, the reason of the error has to be localized – posing yet another challenge. In this work, we proposed a generic error localization methodology which can be applied with any verification method (e.g. assertion checkers, property checker, symbolic executers) available today and which pinpoints the designer to possible error locations in the code. We exemplary realized the proposed methodology on top of a symbolic executer and demonstrated the applicability on benchmarks realizing a traffic collision avoidance system. As shown by these applications, the proposed methodology can significantly aid designers in the process of error localization.

Acknowledgements

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based Stimuli Generation in the SystemC Verification Library," in *Forum on Specification and Design Languages*. Sophia Antipolis, France: IEEE, 2009.

[3] F. Haedicke and H. M. Le and D. Große and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *Proceedings of the Int'l Symp. on SOC*, Tampere, Finland, 2012.

[4] D. Grosse and R. Drechsler, "CheckSyC: an efficient property checker for RTL SystemC designs," in *Proceedings of the Int'l Symp. on Circuits and Systems*, Kobe, Japan, 2005.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the Conf. on Operating systems design and implementation*, San Diego, USA, 2008.

[6] P. Gonzales-de Aledo and S. Pablo, "FramewORk for Embedded System verificaTion," in *Tools and Algorithms for the Construction and Analysis of Systems*, London, UK, 2015.

[7] P. Gonzales de Aledo, N. Prizigoda, R. Wille, R. Drechsler, and P. Sanchez, "Towards a Verification Flow Across Abstraction Levels: Verifying Implementations Against Their Formal Specification," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pp. 475–488, 2017.

[8] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Processings of the DA Conf.*, San Francisco, USA, 2015.

[9] R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow, "FoREnSiC- An Automatic Debugging Environment for C Programs," in *Proceedings of the Int'l Haifa Verification Conf.*, Haifa, Israel, 2012.

[10] M. Jose and R. Majumdar, "Cause Clue Clauses: Error Localization using Maximum Satisfiability," San Jose, USA, 2010.

[11] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *Int'l Conf. on Formal Methods in CAD*, Austin, USA, 2011.

[12] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proceedings of the Symp. on Principles of Programming Languages*, New Orleans, USA, 2003.

[13] D. Große, R. Wille, R. Siegmund, and R. Drechsler, "Contradiction Analysis for Constraint-based Random Simulation," in *Forum on Specification and Design Languages*. IEEE, 2008.

[14] N. Przigoda, R. Wille, and R. Drechsler, "Contradiction Analysis for Inconsistent Formal Models," in *Symposium on Design and Diagnostics of Electronic Circuits and Systems*. IEEE, 2015.

[15] A. Süllflow, R. Wille, G. Fey, and R. Drechsler, "Evaluation of Cardinality Constraints on SMT-based Debugging," in *Int'l Symp. on Multi-Valued Logic*. IEEE, 2009.

[16] M. Weiser, "Program slicing," in *Proceedings of the Int'l Conf. on Software Engineering*, San Diego, USA, 1981.

[17] E. Alves and M. Gligoric and V. Jagannath and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," 2011.

[18] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, 1976.

[19] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, 1976.

[20] H. Yoshida and G. Li and T. Kamiya and I. Ghosh and S. Rajan and S. Tokumoto and K. Munakata and T. Uehara, "KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution," *IEEE Software*, 2017.

[21] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr, "The auspicious couple: Symbolic execution and WCET analysis," *OpenAccess Series in Informatics*, 2013.

[22] D. Kroening, E. Clarke, and K. Yorav, "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking," in *Processings of the DA Conf.*, Anaheim, USA, 2003.

[23] A. Cimatti and I. Narasamdya and M. Roveri, "Software Model Checking SystemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2013.

[24] L. De Moura and N. Bjørner, "Z3: An efficient SMT Solver," *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[25] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the Int'l Symp. on Code Generation and Optimization*, San Jose, USA, 2004.

[26] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2005.

[27] Hutchins, M. and Foster, H. and Goradia, T. and Sotrand, T., "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of the Int'l Conf. on Software Engineering*, Sorrento, Italy, 1994.