(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
6 August 2009 (06.08.2009)

PCT

(10) International Publication Number
WO 2009/095741 A1

(51) International Patent Classification:
G06F 11/36 (2006.01)

(21) International Application Number:
PCT/IB2008/001327

(22) International Filing Date: 1 February 2008 (01.02.2008)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant (*for all designated States except US*): **THE MATHWORKS, INC** [US/US]; 3 Apple Hill Drive, Natick, Massachusetts 01760-2098 (US).

(72) Inventors; and
(75) Inventors/Applicants (*for US only*): **MUNIER, Patrick** [FR/FR]; 100 C Allée Saint-Exupery, F-38330 Montbonnot Saint Martin (FR). **YIN, Haibin** [FR/FR]; Le Piallon, F-38220 Notre Dame de Mesage (FR). **PREVE, Cyril** [FR/FR]; 8 Grand Rue, F-38610 Gieres (FR).

(74) Agent: **COLE, Tony**; Harrity & Harrity, LLP, 11350 Random Hills Road Suite 600, Fairfax, Virginia 22030 (US).

(81) Designated States (*unless otherwise indicated, for every kind of national protection available*): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

(54) Title: SELECTIVE CODE INSTRUMENTATION FOR SOFTWARE VERIFICATION

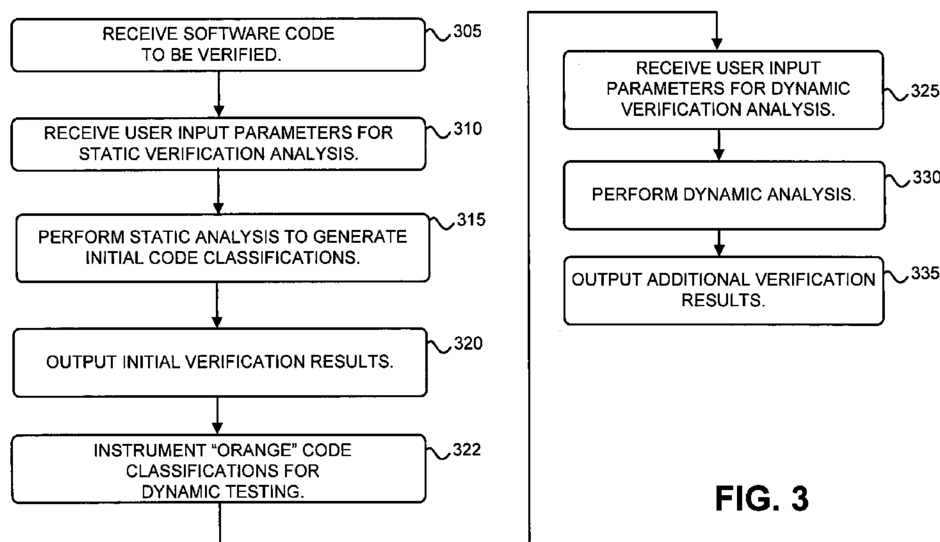


FIG. 3

(57) Abstract: A software testing tool may selectively modify (instrument) portions of a software project for dynamic verification analysis based on the results a static verification analysis. In one implementation, the tool may perform a static verification analysis (315) of the computer code and selectively instrument portions of the computer code for dynamic verification analysis, the selected portion of the computer code including a portion of the computer code that was determined by the static verification analysis to include an unproven error condition. The tool may further execute the computer code to perform a dynamic verification analysis of the selectively instrumented portion of the computer code and store output information describing potential errors identified by the dynamic verification analysis.

WO 2009/095741 A1

SELECTIVE CODE INSTRUMENTATION FOR SOFTWARE VERIFICATION

BACKGROUND

5 **[0001]** Software products can potentially be very large and complex. Software testing is the process used to assess the quality of developed computer software. Quality may be judged based on a number of metrics, such as correctness, completeness, reliability, number of bugs found, efficiency, compatibility, etc.

[0002] The amount of testing required for a particular software project frequently depends on 10 the target for the deployed software. A developer of game software intended for personal computers, for example, may devote relatively little resources into formal testing of the software. In contrast, the developer of a mission critical application in the healthcare, automotive, or utility industry may require a much more rigorous level of software testing.

[0003] One technique for testing software is based on the concept of static verification of the 15 software code. In general, static code verification is an analysis performed without executing the software. Static verification of software code can prove, for example, which operations are free of run-time errors such as numeric overflows, divisions by zero, buffer overflows, or pointer issues, and identify where run-time errors will or might occur.

[0004] In one existing system, static verification is used to classify the code into categories. 20 The categories may include code determined to be good or safe or correct, code determined to have errors, code determined not to be accessible (e.g., "dead code" or "deactivated code") and code for which a determination could not be made (e.g., "don't know"). Code classified as "don't know" represents code that the static verification system could not conclusively determine as including an error. A developer faced with "don't know" code may be required to manually 25 review the code for errors. Manual review of code can be highly labor intensive.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate one or more implementations described herein and, together with the
5 description, explain these implementations. In the drawings:

[0006] Fig. 1 is an exemplary diagram of a system in which concepts described herein may be implemented;

[0007] Fig. 2 is a diagram of an exemplary device corresponding to one of the workstations or servers shown in Fig. 1;

10 **[0008]** Fig. 3 is a flow chart illustrating exemplary operations that may be performed by the verification tool shown in Fig. 1;

[0009] Fig. 4 is a diagram illustrating an exemplary graphical interface in which static verification results may be presented to a user;

[0010] Figs. 5A and 5B are diagrams illustrating exemplary code before and after
15 instrumentation;

[0011] Fig. 6 is a diagram illustrating an exemplary graphical interface in which dynamic verification results may be presented to a user; and

[0012] Fig. 7 is a diagram conceptually illustrating components of a verification tool in an exemplary implementation in which a verification tool is implemented using a client-server
20 model.

DETAILED DESCRIPTION

[0013] The following detailed description refers to the accompanying drawings. The same reference numbers in different drawings may identify the same or similar elements. Also, the

following detailed description does not limit the invention.

OVERVIEW

[0014] Implementations described herein relate to an automated software testing tool in which the results of a static verification analysis technique are used to select portions of the code for additional analysis. The selected portions may include the code determined by the static verification analysis to have an unknown or unproven error condition. These selected portions may be modified (instrumented) for dynamic analysis and then executed in conjunction with input test values to perform a dynamic verification analysis of the code. The results of the dynamic analysis can provide additional verification information about the code that was determined to be unproven by the static verification testing.

DEFINITIONS

[0015] "Static verification analysis," "static verification testing," or "static analysis techniques," as the phrases are used herein, generally refer to an analysis of computer code that is performed without executing the code. For example, static code analysis may examine code using abstract interpretation techniques to verify all possible execution paths of a program.

[0016] "Dynamic verification analysis," "dynamic testing," or "dynamic analysis techniques," as the phrases are used herein, refer to verification of software performed by or during the execution of the software. Dynamic verification may involve, for example, executing the software with a set of test input values.

SYSTEM DESCRIPTION

[0017] Fig. 1 is an exemplary diagram of a system 100 in which concepts described herein may be implemented. The system may include one or more personal computers or workstations 110, one or more servers 120, and a network 130. Consistent with aspects described herein,

software verification tool 105 may be executed by one or more of servers 120 and workstations 110 to assist in software verification.

[0018] Workstations 110 may include computing devices, such as desktop or laptop computers, that may be used for general computing tasks. In general, users of workstations 110 may be software developers. The users may use verification tool 105 to assist in verifying their developed software code. In some implementations, as shown in Fig. 1, verification tool 105 may include client-side components and server-side components. The client-side components may be executed at the user's workstation 110 while the server-side components may execute at one or more of servers 120. In an alternative implementation, and possibly under certain restrictions on use such as the size of the code to be verified, verification tool 105 may execute exclusively at the user's workstation 110.

[0019] In some implementations, workstations 110 may execute a technical computing environment (TCE) that presents a user with an interface that enables efficient analysis and generation of technical applications. For example, the TCE may provide a numerical and/or symbolic computing environment that allows for matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and/or interfacing with programs in other languages. The TCE may be textual and/or graphical.

[0020] Servers 120 may each include a device, such as a computer or another type of computation or communication device, a thread or process running on one of these devices, and/or an object executable by one of these devices. Server device 120 may generally provide services to other devices (e.g., workstations 110) connected to network 130. In one embodiment, one or more of server devices 120 may include server components of software verification tool 105.

[0021] Fig. 2 is a diagram of an exemplary device 200 corresponding to one of workstations 110 or servers 120. As illustrated, device 200 may include a bus 210, a processing unit 220, a main memory 230, a read-only memory (ROM) 240, a storage device 250, an input device 260, an output device 270, and/or a communication interface 280. Bus 210 may include a path that
5 permits communication among the components of workstation 110.

[0022] Processing unit 220 may include a processor, microprocessor, or other types of processing logic that may interpret and execute instructions. Main memory 230 may include a random access memory (RAM) or another type of dynamic storage device that may store information and instructions for execution by processing unit 220. ROM 240 may include a
10 ROM device or another type of static storage device that may store static information and/or instructions for use by processing unit 220. Storage device 250 may include a magnetic and/or optical recording medium and its corresponding drive.

[0023] Input device 260 may include a mechanism that permits an operator to input information to workstation 110, such as a keyboard, a mouse, a pen, a microphone, a touch-
15 sensitive display, voice recognition and/or biometric mechanisms, etc. Output device 270 may include a mechanism that outputs information to the operator, including a display, a printer, a speaker, etc. Communication interface 280 may include any transceiver-like mechanism that enables workstation 110 to communicate with other devices and/or systems. For example, communication interface 280 may include mechanisms for communicating with another device
20 or system via a network.

[0024] As will be described in detail below, workstation 110 may perform certain operations in response to processing unit 220 executing software instructions contained in a computer-readable medium, such as main memory 230. A computer-readable medium may be defined as a

physical or logical memory device. The software instructions may be read into main memory 230 from another computer-readable medium, such as storage device 250, or from another device via communication interface 280. The software instructions contained in main memory 230 may cause processing unit 220 to perform processes that will be described later.
5 Alternatively, hardwired circuitry may be used in place of or in combination with software instructions to implement processes described herein. Thus, implementations described herein are not limited to any specific combination of hardware circuitry and software.

[0025] Although Fig. 2 shows exemplary components of device 200, in other implementations device 200 may contain fewer, different, or additional components than
10 depicted in Fig. 2. In still other implementations, one or more components of device 200 may perform one or more tasks performed by one or more other components of device 200.

SOFTWARE VERIFICATION TOOL

[0026] As previously mentioned, verification tool 105 may be used to measure the quality of developed computer software. Verification tool 105 may perform both static and dynamic
15 software verification. The dynamic software verification may be selectively implemented so that only portions of the code determined to be unproven by the static verification will be instrumented for dynamic analysis.

[0027] In one implementation, verification tool 105 may be used in the context of a technical computing environment. A "technical computing environment," as the term is used herein, is to
20 be broadly interpreted to include any hardware and/or software based logic that provides a computing environment that allows users to perform tasks related to disciplines, such as, but not limited to, mathematics, science, engineering, medicine, business, etc., more efficiently than if the tasks were performed in another type of computing environment, such as an environment that

required the user to develop code in a conventional programming language, such as C++, C, Ada, Java, Javascript, Perl, Ruby, Fortran, Pascal, etc. A technical computing environment may additionally provide mathematical functions and/or graphical tools or blocks (e.g., for creating plots, surfaces, images, volumetric representations, etc.). A technical computing environment, however, is not limited to performing disciplines discussed above, and may be used to perform computations relating to fields of endeavor not typically associated with mathematics, science or business. Verification tool 105 may operate as a component in a technical computing environment to verify code created with the technical computing environment. For example, the technical computing environment may give the user an option to create graphical models. The technical computing environment may then compile the created graphical model for execution on a target system. Verification tool 105 may be used to verify the code that embodies the graphical model. Alternatively, verification tool 105 may be used to verify generated code or the graphical model itself.

[0028] More generally, although verification tool 105 may be used in the context of a technical computing environment, verification tool 105 may be used with any software development project. For example, verification tool 105 may analyze code written in conventional programming language, such as C++, C and Ada, etc., and which is produced manually by a developer with or without the use of a technical computing environment.

[0029] Fig. 3 is a flow chart illustrating exemplary operations that may be performed by verification tool 105.

[0030] Verification tool 105 may initially receive the software code that is to be verified (block 305). For example, a user at one of workstations 110 may use verification tool 105 to select one or more files that contain the software code that is to be verified. The software code

may be textual source code, or code describing a graphical model created using a technical computing environment, or an intermediate representation of the textual and/or graphical source code.

[0031] The user may additionally provide verification tool 105 with input parameters for the software verification (block 310). The input parameters may relate to, for example, options or parameters applicable to a static verification analysis performed by verification tool 105. These static verification options may include, for example, the names of the files or procedures that are to be analyzed, the specific static verification algorithms to use, and/or the options relating to the static verification algorithms that are to be used.

[0032] Verification tool 105 may perform a static verification analysis to generate initial classifications for the code (block 315). As previously mentioned, static verification analysis may involve analysis of the software code without execution of the code. For example, the static verification may involve analysis of the code to construct a model of the code (i.e., an abstract representation of the code). The model can be used for matching commonly occurring error patterns to the code. The model may also be used to perform some kind of data-flow analysis of the code to infer the possible values that variables might have at certain points in the program. Data-flow analysis can be used for vulnerability checking. Static verification analysis may be used to prove which operations are free of run-time errors or to find possible errors. Errors that may be found include: overflows and underflows; divisions by zero and other arithmetic errors; out-of-bounds array access; illegally dereferenced pointers; read-only access to non-initialized data; dangerous type conversions; dead code; access to null pointers; dynamic errors related to object programming and inheritance; errors related to exception handling; and non-initialized class members in the C++ language.

[0033] As a result of the static analysis, verification tool 105 may classify the code into classifications that relate to possible errors in the code. Verification tool 105 may output the verification results from the static analysis (block 320). For example, the results may be saved to a file, shown to a user on a display, stored in a database, etc.

5 [0034] In one implementation, the classification may include classifying each possible failure point in the source code into classes that define: (1) code that has no errors, code that might have errors (unknown or unproven conditions), (2) code that definitely has errors, or (3) code that cannot be reached. The classifications may be presented to the user in a number of possible ways, such as by changing the appearance of the code (e.g., font type, font size, font 10 color, etc.) based on its classification. In one particular implementation, the code may be presented using color codes. For example, the code may be shown to the user as GREEN code (code that has no errors), RED (code that definitely has errors), GRAY (code that cannot be reached), or ORANGE (unknown or unproven error conditions). This color-identifier assignment is for illustration purposes only, and in alternative embodiments, other colors or other 15 visual or non-visual schemes for identifying different types of verified code may be used, as determined by one of skill in the art.

[0035] Fig. 4 is a diagram illustrating an exemplary graphical interface 400 in which verification results may be presented to a user, such as the initial verification results (Fig. 3, block 320). Graphical interface 400 may be presented to a user, for example, by verification tool 20 105 on a display of one of workstations 110.

[0036] Graphical interface 400 may include an entities list section 405 and an indicator list section 410. Indicator list section 410 may include a number of columns (numbered 411-415 in Fig. 4), where each column provides information about the corresponding entity shown in

section 410.

[0037] Entities list section 405 may display the names of the files and underlying functions that have been analyzed. Each file and function may be visually distinguished based on its error classification. In one implementation, as mentioned above, the text of each file may be color coded (e.g., RED, ORANGE, GREEN, or GRAY) to indicate its error classification. In this example, however, the "color" codes are shown using underlined, italic, and bold fonts, in which RED code is shown in italic, GREEN code is shown underlined, and ORANGE code is shown in bold.

[0038] Each file and underlying function may be colorized according to the most critical error found. For example, the file "polyspace_main.c," labeled as file 430, is GREEN, indicating no errors were found. The file "example.c," labeled as file 431, is red, indicating errors were found. In this example, the user has chosen to drill-down into "example.c" by selecting the "+" icon, which shows the functions 432 contained in "example.c". Functions 432 may be similarly color-coded. For example, the function "close_to_zero" is shown as GREEN code, the function "pointer_arithmetic" is shown as RED code, and the function "recursion" is shown as orange code.

[0039] For each file or function, indicator list section 410 may include a column 411 that displays the software reliability of that code, where 100% indicates complete reliability of the code for the code category and 0% means no reliability. Column 412 may indicate the number of RED code segments, column 413 may indicate the number of GRAY segments, column 414 may indicate the number of ORANGE segments, and column 415 may indicate the number of GREEN segments.

[0040] As shown in Fig. 4, the results of a static verification analysis can be concisely

presented to a user in a format in which the user can quickly gauge which code merits further attention.

[0041] In practice, ORANGE (unproven) code tends to be particularly troubling for developers. Because ORANGE code represents an unknown error state, ORANGE code may need to be manually verified by the software developer. For a large project, even if only five or ten percent of the code is classified as ORANGE, the manual effort required to review such code can be extensive.

[0042] Consistent with embodiments described herein, verification tool 105 may automatically instrument ORANGE code for further dynamic verification analysis. The dynamic verification analysis can provide further clarification on whether code initially classified as ORANGE code needs to be manually reviewed. "Instrumenting code," as the phrase is used herein, may generally refer to modifying the code to embed test or verification statements in the code. These statements may be used to monitor or catch conditions that result in errors or other problems or properties of the code. Referring back to Fig. 3, verification tool 105 may instrument the ORANGE code for dynamic testing (block 322).

[0043] Figs. 5A and 5B are diagrams illustrating exemplary code before and after instrumentation. As shown in Fig. 5A, a section of code 500 may include two statements, a GREEN statement 501 and an ORANGE statement 502. Fig. 5B shows code 500 after an exemplary automated code instrumentation procedure. Here, the ORANGE statement 502 is instrumented, while GREEN statement 501 has not been modified. Selective instrumentation of the code in this manner (e.g., only instrumenting ORANGE statements) may reduce the time required to test the code.

[0044] As shown in Fig. 5B, modified statement 502 may include an initial "if" statement

510 that tests whether the denominator (i.e., the variable "K") in Fig. 5A is equal or close to zero. If it is not, the original statement is executed at statement 511. If the denominator is equal or close to zero however, various error or warning messages may be raised. In this example, the warning messages may vary based on flags that define the environment in which the code is
5 executing. Whether an error is generated may depend on the values of the input test cases. That is, an error will be generated only when the input test cases cause the variable K to be zero when statement 502 is executed.

[0045] Referring back to Fig. 3, the user may additionally provide verification tool 105 with input parameters for dynamic verification of the software (block 325). The input parameters
10 may relate to, for example, options or parameters applicable to a dynamic verification analysis performed by verification tool 105.

[0046] A particular type of input parameter may define ranges or values used for "test cases" for the software. The test cases may be applicable to software that receives external values during execution. For example, the software may be designed to act on input received from
15 users, external sensors, or external applications or computing devices. The variables needed to describe the external stimuli may be used to define the test cases. The user may define a range that is to be covered for each variable to define all of the test cases. For example, considered a software system in which two integer variables, X and Y, represent all of the external inputs received by the system. The designer may decide that a satisfactory test can be obtained when X
20 is constrained to the range of zero to five and Y is constrained to the range of zero to two. Further, the designer may decide that 10 dynamic tests should be played. The complete set of (X,Y) test cases for this example system would be [(0,0), (0,1), (0,2), (1, 0), (1,1), (1,2), (2,0) … (5, 2)]. Of these, verification tool 105 may randomly select 10 of these pairs of values, which

may be used as the set of test cases that are used to test the software.

[0047] Other techniques for specifying test cases may be used. For example, instead of having test cases automatically generated based on a range of variable values entered by the user, the user may specify specific test cases to be used, such as by manually entering the specific test
5 cases. Additionally, in some implementations, test cases generated by multiple different sources may be combined or imported for use in the final set of test cases. This can potentially provide the advantage of improving the structural coverage and over all quality of the dynamic testing.

[0048] Verification tool 105 may next perform the dynamic verification analysis (block 330). The dynamic verification analysis may be performed by compiling the code and then executing it
10 multiple times, with each execution using a different test case from the set of defined test cases.

[0049] As a result of the dynamic analysis, verification tool 105 may output additional verification results that provide additional verification information about the code that has been instrumented for dynamic verification analysis (block 335). The additional verification information can be used to reduce the number of "false positives" (i.e., uncertain output
15 conditions) generated by the initial static verification analysis.

[0050] Fig. 6 is a diagram illustrating an exemplary graphical interface 600 in which the additional verification results may be presented to a user. Graphical interface 600 may be presented to a user, for example, by verification tool 105 on a display of a workstation 110.

[0051] Graphical interface 600 may include a variable list section 605, a test summary
20 section 610, and an error log section 615. Variable list section 605 may contain information about each of the variables used to define the test cases. As shown, variable list section 605 may include a variable name field 620, a variable type field 625, and a variable values field 630. Each item in variable name field 620 may be the name of an input variable of the software, or an

output parameter of an external function that is stubbed by verification tool 105. Variable type field 625 may list the type of each variable. For example, the variable returned from the function "random_float" is a variable of type "float32." Variable values field 630 may list the range of values, such as the minimum and maximum value that was assigned to the variable.

5 **[0052]** Test summary section 610 may provide summary information or configuration information relating to the dynamic verification analysis. As shown in Fig. 6, the summary information may include a configuration section 640 and a results section 645. In the configuration section 640, the user may enter configuration information relating to the dynamic analysis. For example, the user may enter the number of tests to be executed, the number of 10 loops to run before determining that a loop is an infinite loop, and the timeout period associated with each test. Results section 645 may display results relating to the dynamic verification analysis. In one implementation, the results may be dynamically updated as the dynamic verification analysis executes. The displayed results may include, for example, the number of completed tests, the number of tests in which no run-time error was detected in instrumented 15 code sections, and the number of test in which run-time errors were detected in instrumented code sections.

[0053] Error log section 615 may provide a detailed list of all of the errors that occurred during the dynamic verification analysis. For example, as shown in Fig. 6, for each error, error log section 615 may display the name of the file in which the error occurred, the line number and 20 column of the error, a description of the error, and the number of test cases in which the error occurred.

[0054] Through graphical interface 600, a user may monitor and/or control the dynamic verification analysis performed on selected code. Based on the dynamic verification results, the

user can increase their confidence in the reliability of the code. As the tests performed may not be exhaustive, they may not definitively prove that the code is run-time error free. But a set of performed tests without any run-time errors may increase the user's confidence in the code's reliability.

5 **[0055]** In one implementation, based on the results of the dynamic verification analysis, verification tool 105 may provide results of the dynamic verification analysis in conjunction with or together with the color codes generated by the static verification analysis. For example, one or more additional categories may also be created. For example, a "DYNAMIC VERIFICATION GREEN" category may be created, which may be associated with a different 10 color, such as a shade of green different than the green used to present the static verification analysis GREEN code. In this manner, the user can easily distinguish between the GREEN code generated by the static verification analysis and the "GREEN" code generated by the dynamic verification analysis. The categories for the dynamic verification and static verification may be presented using an interface similar to that shown in Fig. 4.

15 **[0056]** Fig. 7 is a diagram conceptually illustrating components of verification tool 105 in an exemplary implementation in which verification tool 105 is implemented using a client-server model.

[0057] As shown in Fig. 7, verification tool 105 may include components 710 that execute on a server, such as one or more of servers 120, and components 715 that execute on a client, 20 such as one of workstations 110. More particularly, server components 710 may include static code verification component 711 and results 712. Client components 715 may include source code 720, result viewer component 721, potential bug list 722, test case generator component 723, test cases 724, dynamic instrumentation component 725, instrumented code 726, executable

component 727, and potential bug list 728. Client components 715 may be associated with, for example, one of workstations 110.

[0058] In operation, a user may load or otherwise generate source code 720. The source code may be transmitted to server 120 via a network, such as network 130. Server 120 may
5 function as a static code verification component for a number of users, such as for all of the software developers for a particular company.

[0059] Static code verification component 711 may analyze the source code using static verification analysis techniques, such as using the techniques discussed previously with respect to block 315 (Fig. 3). The initial verification results may be stored as results 712. Results 712
10 may be transmitted back to client 110 for viewing and/or analysis.

[0060] Client components 715 may particularly include result viewer 721. Result viewer 721 may present results 722 to the user. Result viewer 721 may, for example, contain a graphical user interface, such as the one shown in Fig. 4, that uses color-codes to illustrate the error status of different sections of code. Result viewer 721 may additionally output or store
15 results 712 as potential bug list 722.

[0061] Client components 715 may additionally include test case generator component 723 to generate test cases 724. Test case generator component 723 may, for example, present an interface such as variable list section 605 of graphical interface 600 (Fig. 6). Through this graphical interface, users may define, for instance, value ranges that are to be tested for each
20 input variable of the software. Based on this information, test case generator component 723 may generate test cases 724.

[0062] Client components 720 may additionally include dynamic instrumentation component 725 to instrument source code 720 based on results 712. As previously discussed, this

instrumentation may be performed such that code segments that were determined by the static verification analysis to have an unknown or unproven error condition (e.g., ORANGE code) may be instrumented. Source code 720, after instrumentation by dynamic instrumentation component 725, is stored as instrumented code 726.

5 **[0063]** Instrumented code 726 may be compiled and run in conjunction with test cases 724 as executable component 727 and may generate potential bug list 728 during the execution.

[0064] Although Fig. 7 illustrates an implementation of verification tool 105 using a client-server model, in an alternative implementation, all or substantially all of the functionality of verification tool 105 may be implemented on a single computer, such as a single workstation

10 110.

CONCLUSION

[0065] Techniques were described herein for testing code using both static and dynamic verification analysis techniques. The dynamic verification analysis techniques may be automatically applied to selected portions of the code. The selected portions of the code may

15 correspond to portions of the code that was determined to correspond to unproven sections of the code.

[0066] The foregoing description of implementations provides illustration and description, but is not intended to be exhaustive or to limit the invention to the precise form disclosed. Modifications and variations are possible in light of the above teachings or may be acquired from

20 practice of the invention.

[0067] For example, while a series of acts has been described with regard to Fig. 3, the order of the acts may be modified in other implementations. Further, non-dependent acts may be performed in parallel.

[0068] Also, the term “user” has been used herein. The term “user” is intended to be broadly interpreted to include a workstation or a user of a workstation.

[0069] It will be apparent that embodiments, as described herein, may be implemented in many different forms of software, firmware, and hardware in the implementations illustrated in
5 the figures. The actual software code or specialized control hardware used to implement embodiments described herein is not limiting of the invention. Thus, the operation and behavior of the embodiments were described without reference to the specific software code--it being understood that one would be able to design software and control hardware to implement the embodiments based on the description herein.

10 [0070] Further, certain portions of the invention may be implemented as "logic" that performs one or more functions. This logic may include hardware, such as an application specific integrated circuit or a field programmable gate array, software, or a combination of hardware and software.

[0071] No element, act, or instruction used in the present application should be construed as
15 critical or essential to the invention unless explicitly described as such. Also, as used herein, the article "a" is intended to include one or more items. Where only one item is intended, the term "one" or similar language is used. Further, the phrase "based on" is intended to mean "based, at least in part, on" unless explicitly stated otherwise.

WHAT IS CLAIMED IS:

1. A computing device-implemented method, comprising:

receiving computer code;

performing a static verification analysis of the computer code;

5 selectively instrumenting a portion of the computer code for dynamic verification analysis, the selected portion of the computer code including a portion of the computer code that has been determined by the static verification analysis to include an unproven error condition;

executing the computer code to perform a dynamic verification analysis of the selectively instrumented portion of the computer code; and

10 storing output information describing potential errors identified by the dynamic verification analysis.

2. The computing device-implemented method of claim 1, wherein the static verification analysis detects one or more of errors relating to one or a combination of the 15 following: overflows and underflows; divisions by zero; out-of-bounds array access; illegally dereferenced pointers; read-only access to non-initialized data; dangerous type conversions; dead code; or access to null pointers.

3. The computing device-implemented method of claim 1, wherein the computer 20 code is classified, based on the static verification analysis, into classes that include code that has an unproven error condition, code that has no errors, and code that is known to have errors.

4. The computing device-implemented method of claim 3, wherein the classes

additionally include a class for unreachable code.

5. The computing device-implemented method of claim 3, further comprising:

presenting the output information using color codes to represent the classifications of the computer code.

6. The computing device-implemented method of claim 1, further comprising:

generating test cases that define sets of values for variables used by the computer code; and

performing the dynamic verification analysis using the test cases.

7. The computing device-implemented method of claim 6, where generating the test cases includes automatically generating the test cases based on input parameters or manually entering test cases.

8. The computing device-implemented method of claim 1, wherein the computer code is code manually generated by a developer or automatically generated based on a model created in a technical computing environment .

9. A computer-readable medium containing programming instructions for execution by one or more processing devices, the computer readable medium comprising:

instructions to receive computer code;

instructions to perform a static verification analysis of the computer code;

instructions to classify, based on the static verification analysis, the computer code into diagnostic categories that relate to reliability of the computer code;

instructions to instrument portions of the computer code for a dynamic verification analysis, the instrumented portions corresponding to computer code that was categorized into
5 selected diagnostic categories;

instructions to perform the dynamic verification analysis on the computer code to identify potential errors in the computer code; and

instructions to store output information describing the identified potential errors.

10 10. The computer-readable medium of claim 9, wherein the static verification analysis detects one or more of errors relating to overflows and underflows; division by zero; out-of-bounds array access; illegally dereferenced pointers; read-only access to non-initialized data; dangerous type conversions; dead code; or access to null pointers.

15 11. The computer-readable medium of claim 9, wherein the instructions to instrument portions of the computer code further include:

instructions to instrument portions of the computer code that were categorized into a diagnostic category that includes code having an unproven error condition.

20 12. The computer-readable medium of claim 9, further comprising:

instructions to present the output information using color codes to represent the diagnostic categories.

13. The computer-readable medium of claim 9, wherein the computer code represents code generated based on a model created in a technical computing environment.

14. A computing device comprising:

5 a component to receive classifications for computer code that is based on potential error conditions, identified via static verification analysis techniques, in the computer code;

a dynamic instrumentation component to selectively instrument portions of the computer code for dynamic verification of the computer code, the portions of the computer code selected for instrumentation being based on the classifications of the computer code;

10 a test case generator to generate test cases for input to the computer code; and

an execution component to execute the selectively instrumented computer code in conjunction with the test cases to perform a dynamic verification analysis of the computer code.

15. The device of claim 14, where the dynamic instrumentation component

15 selectively instruments the portions of the computer code that is classified as having an unknown error condition.

16. The device of claim 14, where the potential error conditions include code that has an has an unproven error condition, code that has no errors, and code that is known to have

20 errors.

17. The device of claim 14, where the computing device additionally presents output information describing potential errors identified by the dynamic verification analysis.

18. The device of claim 17, where the output information uses color codes to represent the classifications of the computer code.

5 19. The device of claim 14, wherein the computer code represents code generated based on a model created in a technical computing environment.

10

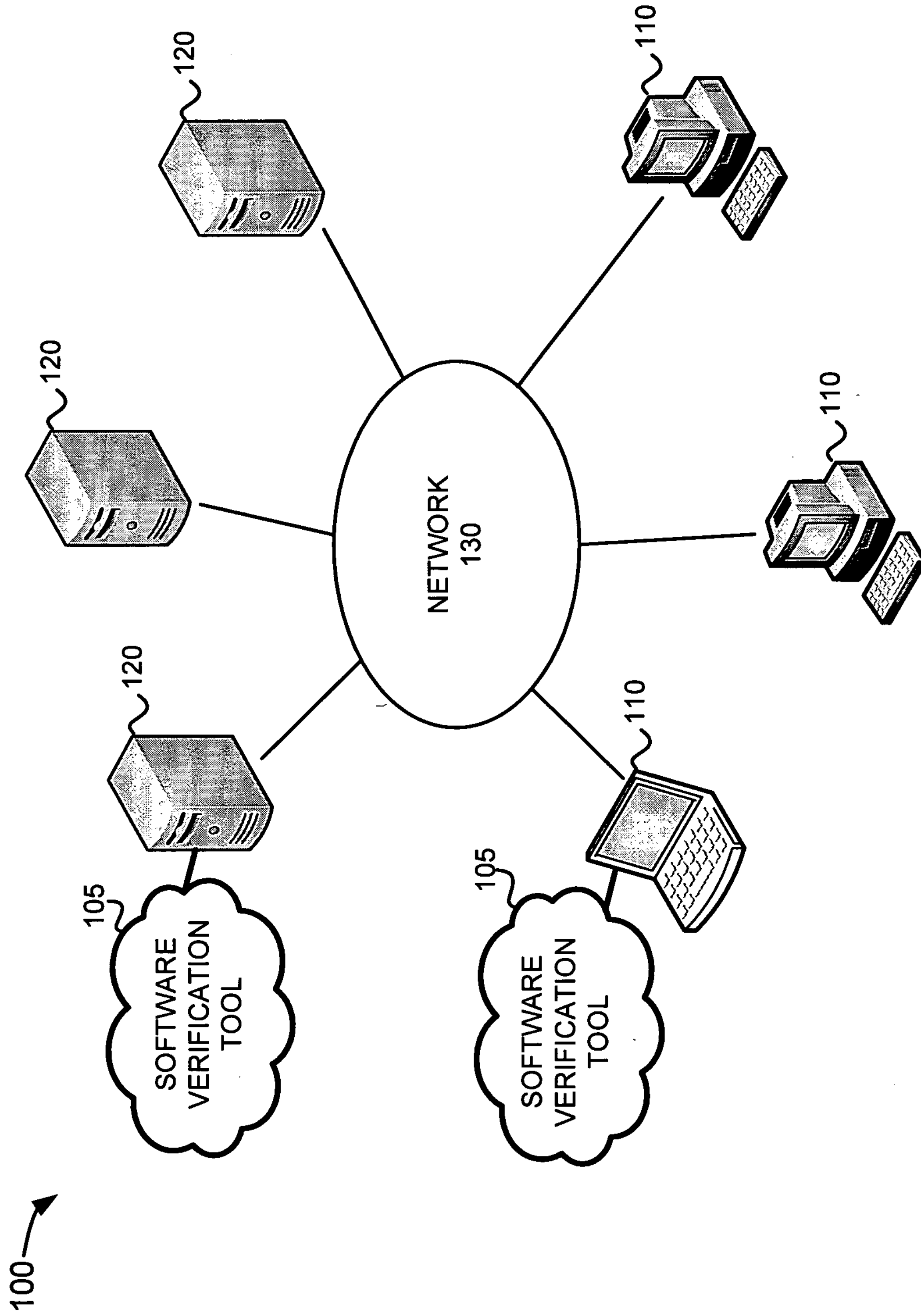


FIG. 1

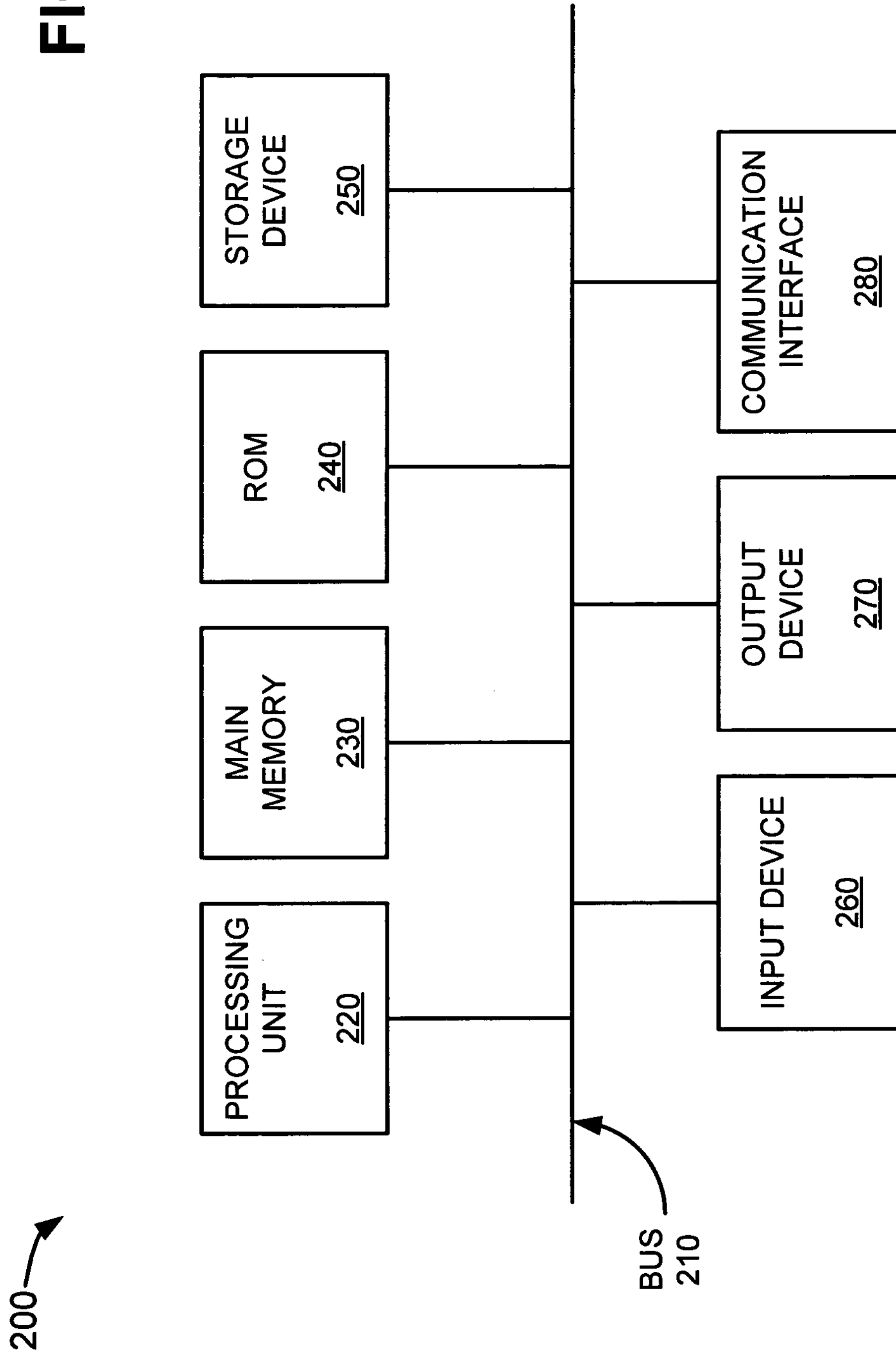


FIG. 2
200
PROCESSING UNIT 220
MAIN MEMORY 230
ROM 240
STORAGE DEVICE 250
BUS 210
INPUT DEVICE 260
OUTPUT DEVICE 270
COMMUNICATION INTERFACE 280

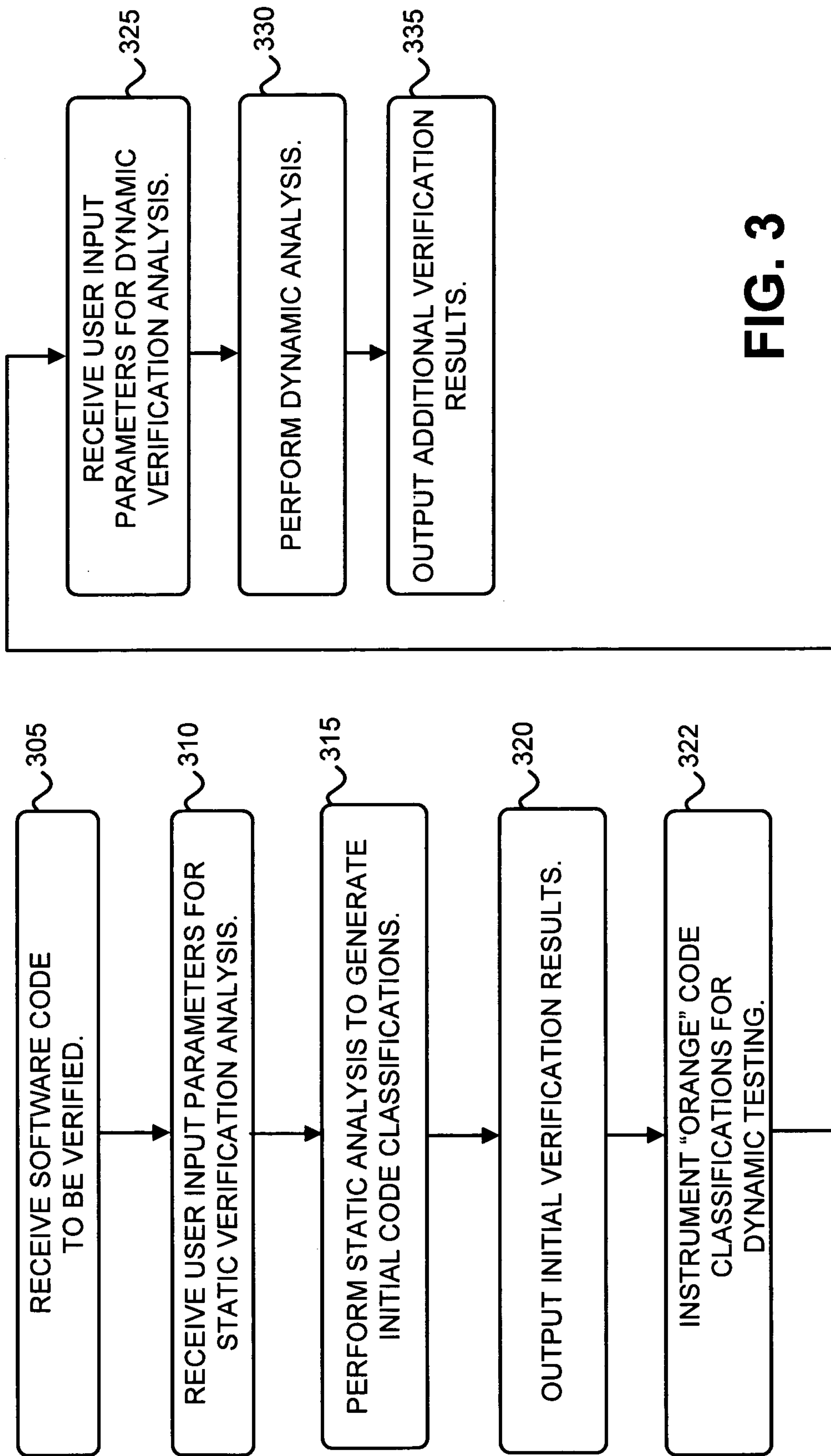


FIG. 3

400

ENTITIES	%	!	X	?	✓
polyspace_main.c	0				
example.c	91	4	1	3	26
close_to_zero()	100				3
non_infinite_loop()	100				4
pointer_arithmetic()	88	1	1	1	5
RTE()	100	1			3
recursion()	80			1	4
recursion_caller()	100	1			2
square_root()	100	1			1
square_root_conv()	100				2
unreachable_code()	100				2
get_oil_pressure()	0			1	

430 431 432 405 410 411 412 413 414 415

KEY:
GREEN CODE – UNDERLINED
RED CODE – ITALIC
ORANGE CODE- BOLD

FIG. 4

500

```
501 Y = X / (X-U)    /* GREEN */
502 Z = Y / K
```

FIG. 5A

```
    Y = X / (X-U)    /* GREEN */
510 IF (K!=0.0)
    {
        Z = Y/K   511
    }
    ELSE {
      #IFDEF RAPID_PROTOTYPING
       /* PRINT RUN TIME ERROR, WARNING
        MESSAGE, ETC.  */
      #ELSEIF PRODUCTION
       /* PRODUCTION ERROR INDICATION */
       ASSERT(DIV_BY_ZERO)
    }
```

502

FIG. 5B

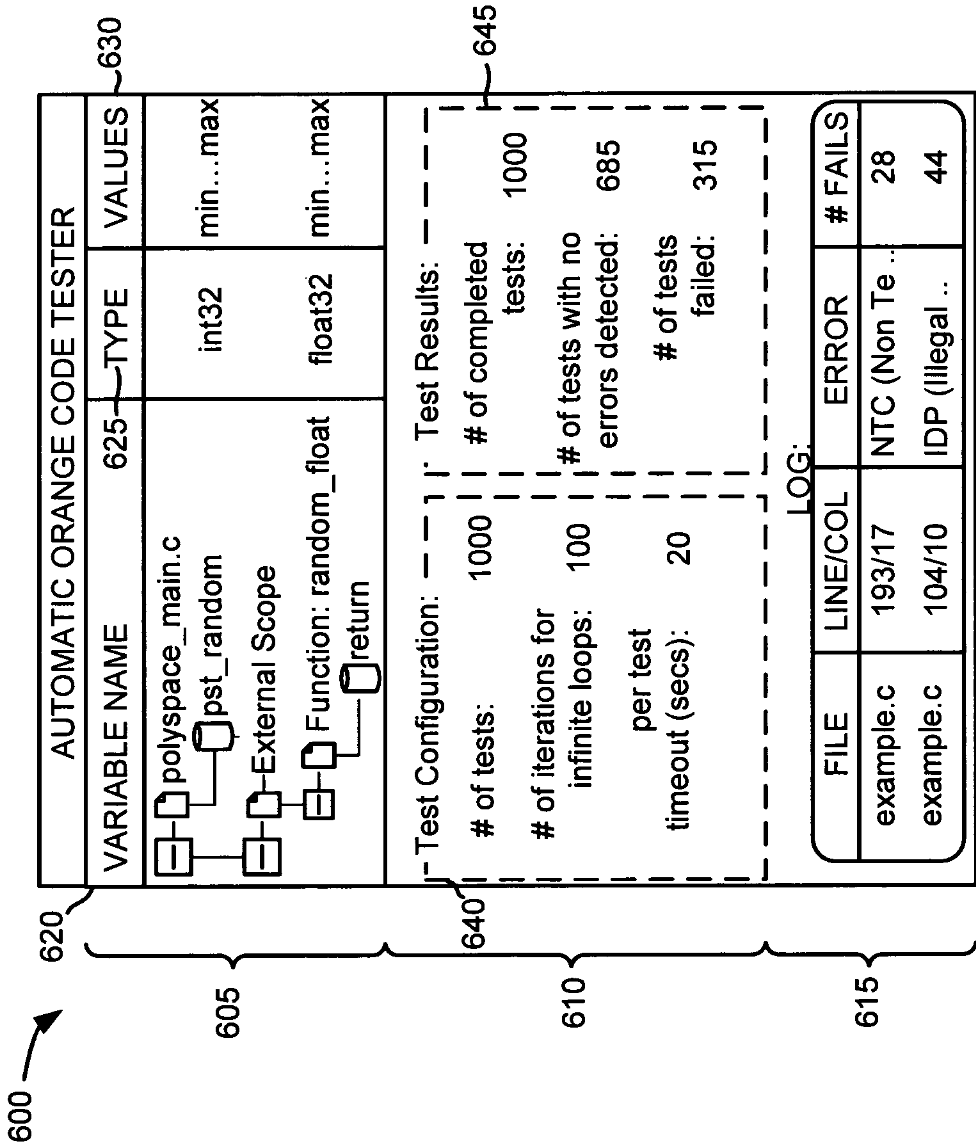
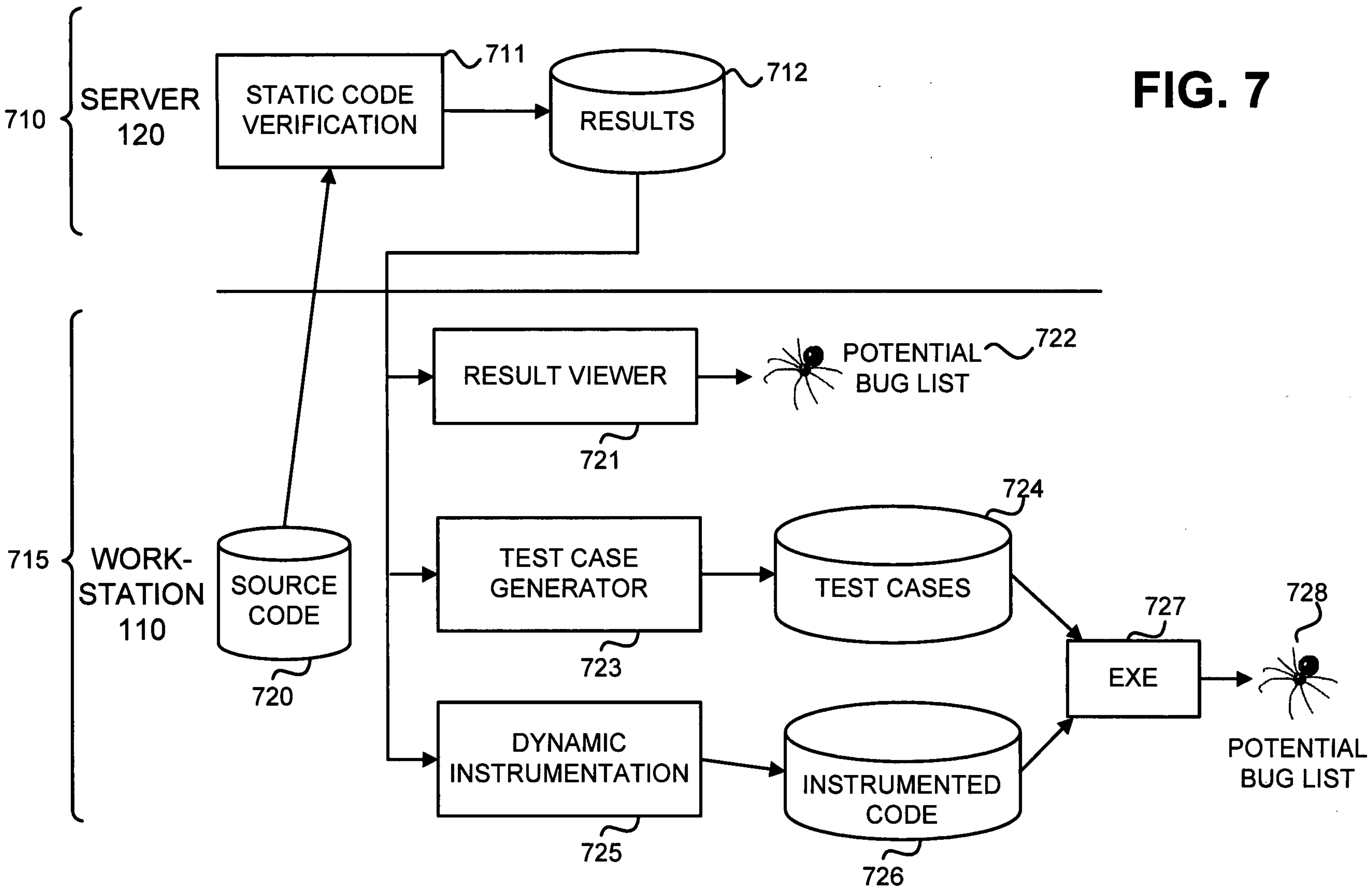


FIG. 6


710
SERVER
120
STATIC CODE
VERIFICATION
711
RESULTS
712
FIG. 7
715
WORK-
STATION
110
SOURCE
CODE
720
RESULT VIEWER
721
POTENTIAL
BUG LIST
722
TEST CASE
GENERATOR
723
TEST CASES
724
DYNAMIC
INSTRUMENTATION
725
INSTRUMENTED
CODE
726
EXE
727
728
POTENTIAL
BUG LIST

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2008/001327

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F11/36

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	ERKKINEN T., HOTE C.: "Automatic Flight Code Generation with Integrated Static Run-Time Error Checking and Code Analysis" 2006 AIAA MEETING PAPERS, vol. 11, no. 19-20, 2006, pages 1-9, XP007905828 US the whole document -----	1-19
X	C. DEL GROSSO G. ANTONIOL M. DI PENTA: "An evolutionary testing approach to detect buffer overflow" PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM OF SOFTWARE RELIABILITY ENGINEERING (ISSRE), 2004, pages 77-78, XP007905817 St Malo, Bretagne, France the whole document -----	1,2, 6-11, 13-15, 17-19

-/--

[X] Further documents are listed in the continuation of Box C.

[ ] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
13 October 2008	20/10/2008

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Salsa, Francesco

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2008/001327

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	DEUTSCH A.: "Static verification of dynamic properties" WHITE PAPER, POLYSPACE TECHNOLOGIES, 27 November 2003 (2003-11-27), XP007905827 the whole document	1-19
A	BRAT G ET AL: "Precise and Scalable Static Program Analysis of NASA Flight Software" AEROSPACE, 2005 IEEE CONFERENCE BIG SKY, MT, USA 05-12 MARCH 2005, PISCATAWAY, NJ, USA,IEEE, PISCATAWAY, NJ, USA, 5 March 2005 (2005-03-05), pages 1-10, XP031213684 ISBN: 978-0-7803-8870-3 the whole document	1-19
A	DING Z ET AL: "Practical Strategies to Improve Test Efficiency" TSINGHUA SCIENCE AND TECHNOLOGY, TSINGHUA UNIVERSITY PRESS, BEIJING, CN, vol. 12, 1 July 2007 (2007-07-01), pages 250-254, XP022933117 ISSN: 1007-0214 [retrieved on 2007-07-01] the whole document	1-19
A	BEIZER: "SOFTWARE TESTING TECHNIQUES, 1990, pages 155-157, XP008097104 USA the whole document	1-19