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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
23 October 2014 (23.10.2014)

WIPO | PCT

(10) International Publication Number
WO 2014/172001 A1

(51) International Patent Classification:
G01R 31/28 (2006.01)

(21) International Application Number:
PCT/US2014/015963

(22) International Filing Date:
12 February 2014 (12.02.2014)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
13/864,191 16 April 2013 (16.04.2013) US

(71) Applicant *(for all designated States except US)*: **ADVANTEST CORPORATION** [JP/JP]; 1-32-1 Asahi-Cho, Nerima-ku, Tokyo, 179-0071 (JP).

(72) Inventors; and

(71) Applicants *(for US only)*: **ELSTON, Mark** [US/US]; 1777 Merlot Way, Salinas, CA 93906 (US). **CHEN, Leon** [US/US]; 6666 Rainbow Dr., San Jose, CA 95129 (US). **SINGH, Harsanjeet** [US/US]; 100 Balceta Ct., Danville, CA 94526 (US). **MAEDA, Hironori** [JP/JP]; 2-11-18 Shinjuku, Tatebayashi-Shi Gumma, 374-0026 (JP). **PRAMANICK, Ankan** [US/US]; 5839 Capilano Dr., San Jose, CA 95138 (US). **KATSU, Youbi** [JP/JP]; 675-14 Takabayashi Minami Machi, Ota, Gunma Prefecture, 3730827 (JP).

(74) Agent: **NANJI, Furqan, A.**; Murabito, Hao & Barnes, LLP, 2N. Market St., 3rd Floor, San Jose, CA 95113 (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, BN, BR, BW, BY, BZ, CA, CH, CL, 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, IR, IS, JP, KE, KG, 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, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, 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, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

(54) **Title**: IMPLEMENTING EDIT AND UPDATE FUNCTIONALITY WITHIN A DEVELOPMENT ENVIRONMENT USED TO COMPILE TEST PLANS FOR AUTOMATED SEMICONDUCTOR DEVICE TESTING

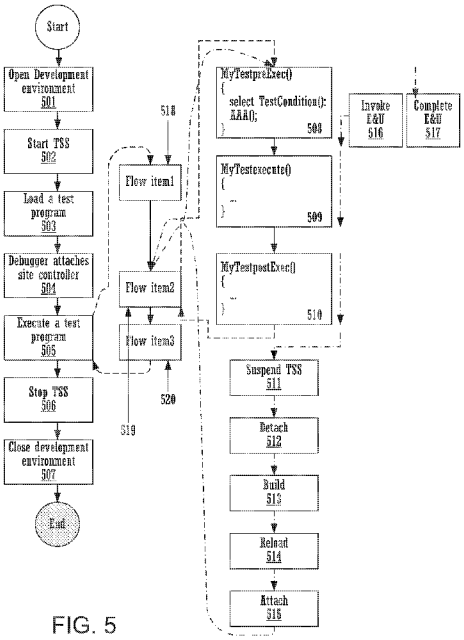

FIG. 5

(57) **Abstract**: A method for debugging test procedures for automated device testing is disclosed. The method comprises receiving a command to update at least one modified test procedure modified during a first debugging session and saving state information for a test plan, wherein the state information comprises information regarding a breakpoint entry location, and wherein the modified test procedure is invoked within the test plan. The method subsequently comprises suspending execution of the test plan and unloading the modified test procedure. It also comprises compiling the modified test procedure to produce a compiled file and then reloading the test procedure into the test plan using the compiled file. Finally, it comprises resuming execution of the modified test procedure in a second debugging session and breaking the execution during the second debugging session at a breakpoint corresponding to the breakpoint entry location.

WO 2014/172001 A1

IMPLEMENTING EDIT AND UPDATE FUNCTIONALITY WITHIN A DEVELOPMENT ENVIRONMENT USED TO COMPILE TEST PLANS FOR AUTOMATED SEMICONDUCTOR DEVICE TESTING

FIELD OF THE INVENTION

[0001] Embodiments of the present invention relate generally to automated device testing and more specifically to a user-friendly and efficient method and system for debugging test procedures for automated device testing.

BACKGROUND OF THE INVENTION

[0002] Automated test equipment (ATE) can be any testing assembly that performs a test on a device, semiconductor wafer or die, etc. ATE assemblies may be used to execute automated tests that quickly perform measurements and generate test results that can then be analyzed. An ATE assembly may be anything from a computer system coupled to a meter, to a complicated automated test assembly that may include a custom, dedicated computer control system and many different test instruments that are capable of automatically testing electronics parts and/or semiconductor. Automatic Test Equipment (ATE) is commonly used within the field of electrical chip manufacturing. ATE systems both reduce the amount of time spent on testing devices to ensure that the device functions as designed and serve as a diagnostic tool to determine the presence of faulty components within a given device before it reaches the consumer.

[0003] In testing devices or products, e.g. after production, it is crucial to achieve among others a high product quality, an estimation of the device or product performance, a feedback concerning the manufacturing process and finally a high customer contentment. Usually a plurality of tests is performed in order to ensure the correct function of a device or product. Because the testing process is expensive, in addition to ensuring performance of the devices, it is also important to maximize testing speed and throughput.

[0004] A test plan or test program comprises all user-defined data and control flows that are necessary to perform a semiconductor device test on an ATE system. More specifically, a test plan is a program where test conditions which satisfy device specifications, combinations of test conditions and test algorithms, and a test flow are described in the language of the tester software. The main control flow in a test program, which dictates the sequence of individual tests to be applied to the DUTs, and the order in which the tests will be applied (which is dependent on the results of individual tests), is referred to as the test program flow.

[0005] When a test program is written for a DUT, in general a predominant part of the program is "data" for device test, and the rest is the program code, which realizes the test methodology itself. The data is DUT-dependent (e.g., power supply conditions, signal voltage conditions, timing conditions, etc.). The test code may consist of methods to load the specified device conditions on to tester hardware, and also those needed to realize the user specified objectives such as data-logging, etc. The ATE framework can also provide a hardware-independent test and tester object model that allows the user to perform the task of DUT test programming.

[0006] Since test methodology is a significant component in device test quality and tester productivity, it is often encapsulated from the end users. This separation results in safer operation of the system. Further, to increase the reusability of test code, such code should be independent of any device-specific data, e.g., pin name, stimulus data, etc. or device-test-specific data, e.g., conditions for DC units, measurement pins, number of target pins, name of pattern file, addresses of pattern programs, etc. If code for a test is compiled with data of these types, the reusability of the test code would decrease. Therefore, any device-specific data or device-test-specific data should be made available to the test code externally as inputs during code execution time.

[0007] In conventional ATE tester software, a "test class," realizes the separation of test data and code (and, hence, the reusability of code) for a particular type of test. Such a test class could be regarded as a "template" for separate instances of a test, which differ from each other only on the basis of device-specific and/or device-test-specific data. Test classes are typically specified in the test plan file. Each test class typically implements a specific type of device test or setup for device test. For example, the tester software can provide sample test classes to

implement functional tests, AC parametric tests, and DC parametric tests. Accordingly, a test class is the template for a test. Device-dependent test conditions are not described in the test class itself. Only an algorithm for executing tests using the test system is described.

[0008] Test classes allow the user to configure class behavior by providing parameters that are used to specify the options for a particular instance of that test. For example, a test class for implementing a functional test may provide parameters to specify a test pattern to execute on the DUTs being tested. Also it may provide parameters to specify the level and timing conditions for the test. Specifying different values for these parameters allows the user to create different instances of the functional test. In one embodiment, test classes are developed in a conventional programming language such as C++.

[0009] A tester software system such as Advantest Corporation's T2000 System Software, typically comprises a compiler, which compiles test plans written in the programming language of the tester system, e.g. OPENSTAR Test Programming Language ("OTPL"). OTPL can be used to describe test execution sequence and the corresponding test conditions used. While OTPL can describe a test plan, it typically does not describe any test algorithm. The test algorithm in a system such as T2000, for example, can be described using the C++ language in a test class as discussed above.

[0010] Most users of an ATE use a development environment, e.g. Microsoft Developers Studio ("MDS") to prepare their test classes. Typically, the users will be provided with plug-ins from the developers of the tester software system so they can edit and create test plans within the same environment. Thus, users are able to write out test classes and test plans and carry out debugging from within the development environment of their choice. A development environment such as MDS can typically provide an "Edit and Continue" feature for debugging projects. "Edit and continue" is a time-saving feature that enables a user to make changes to the source code while the program is in break mode. When the execution of the program is resumed by choosing an execution command such as "continue" or "step," the edit and continue" operation automatically applies the code changes with certain limitations. This allows the user to make changes to the code during a debugging session, instead of having to stop, recompile the entire program, and restart the debugging session.

[0011] The "edit and continue" feature available in typical development environments, e.g., MDS, have several limitations, however, that restrict a user's ability to fully take advantage of the feature. First, it cannot be used in connection with all types of code. For example, it cannot be used in a debugging environment when debugging 64-bit code. As a result, many users of tester software that develop 64-bit test classes and test plans for performance reasons are unable to take advantage of this feature.

[0012] More importantly, the conventional "edit and continue" feature has various limitations on the type of code changes allowed. For example, one drawback is that conventional development environments only allow relatively minor code changes to be made when using the feature. Typically, the altered code section is recompiled, and a small portion of executing memory corresponding to the altered code is swapped out and replaced with a new block of memory generated using the new code. Thereafter, the appropriate instruction pointers are set accordingly.

[0013] The feature, as designed in conventional systems, is not able to handle any intricate code alterations that would require anything more complex than making a minor change in the code, e.g. syntactical changes, which would alter only a relatively small section of loaded memory. As a result, it is not very useful for developers of test classes for ATE. If users make any substantial error in the preparation of the test class or need to replace a test class for any other reason while a test plan is executing, they typically have to stop the test plan and unload it first. Subsequently, the developers need to edit, recompile, and relink the test class. Then, they need to reload the test plan and restart it from the beginning before getting back to the original point in the code. Such a work flow is time-consuming and error-prone.

BRIEF SUMMARY OF THE INVENTION

[0014] Accordingly, what is needed is an automated method and system to allow users to make changes to test classes during execution of a test plan without needing to manually perform all the steps that are currently required when making a change to test classes during debugging, such as unloading and reloading the entire test plan. This allows developers to prepare and debug test classes in a significantly more efficient and less error-prone manner than they were allowed to using conventional systems.

[0015] In order to accomplish this objective, embodiments of the present invention provide edit and update functionality within tester software systems. While debugging, a user may set a breakpoint in the test class source code causing the system to stop. The user can edit the source code, thereby initiating an edit and update session. The state information is then stored and the edited test classes are unloaded. Subsequently, the edit and update procedure will recompile the edited version of the test class and swap the original version with the newly compiled version. The edited test class is then re-executed and automatically stepped back to the same breakpoint as where the user left off debugging and the previously stored state information is restored. The test plan can then continue with the newly edited code.

[0016] A method for debugging test procedures for automated device testing is disclosed. The method comprises receiving a command to update at least one modified test procedure modified during a first debugging session and saving state information for a test plan, wherein the state information comprises information regarding a breakpoint entry location, and wherein the modified test procedure is invoked within the test plan. The method subsequently comprises suspending execution of the test plan and unloading the modified test procedure. It also comprises compiling the modified test procedure to produce a compiled file and then reloading the test procedure into the test plan using the compiled file. Finally, it comprises resuming execution of the modified test procedure in a second debugging session and breaking the execution during the second debugging session at a breakpoint corresponding to the breakpoint entry location, thereby, restoring the state information saved earlier.

[0017] In another embodiment, a computer-readable storage medium is disclosed. The computer-readable storage medium has stored thereon, computer executable instructions that, if executed by a computer system cause the computer system to perform a method for debugging test procedures for automated device testing. This method comprises receiving a command to update at least one modified test procedure modified during a first debugging session and saving state information for a test plan, wherein the state information comprises information regarding a breakpoint entry location, and wherein the modified test procedure is invoked within the test plan. The method subsequently comprises suspending execution of the test plan and unloading the modified test procedure. It also comprises compiling the modified test procedure to produce a compiled file and then reloading the test procedure into the test plan using the compiled file. Finally, it comprises resuming execution of the modified test procedure in a second debugging

session and breaking the execution during the second debugging session at a breakpoint corresponding to the breakpoint entry location, thereby, restoring the state information saved earlier.

[0018] In another embodiment, a system for debugging test procedures for automated device testing is disclosed. The system comprises a memory comprising a development environment stored therein, wherein the development environment is operable to debug a test program, and wherein the development environment comprises a debugger. The system also comprises a processor coupled to the memory, the processor being configured to operate in accordance with the development environment to: (a) save state information for a test plan, wherein the state information comprises information regarding a breakpoint entry location, and wherein a modified test procedure is invoked within the test plan; (b) suspend execution of the test plan; (c) unload the modified test procedure from the test plan; (d) compile the modified test procedure, wherein the compile generates a compiled file associated with the modified test procedure; (e) reload the modified test procedure into the test plan using the compiled file; (f) resume execution of the modified test procedure in a debugging session; and (g) breaking the execution during the debugging session at a breakpoint corresponding to the breakpoint entry location. The state information saved earlier is subsequently restored.

[0019] The following detailed description together with the accompanying drawings will provide a better understanding of the nature and advantages of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0020] Embodiments of the present invention are illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements.

[0021] Figure 1 is an exemplary computer system on which embodiments of the automated test system of the present invention can be implemented in accordance with one embodiment of the present invention.

[0022] Figure 2A is a schematic block diagram for an exemplary automated test equipment apparatus on which embodiments of the present invention can be implemented in accordance with one embodiment of the present invention.

[0023] Figure 2B is a more detailed schematic block diagram of one embodiment of the automated test equipment apparatus of Figure 2A.

[0024] Figure 3 illustrates a diagram regarding the manner in which different test instances can be created from a single test class.

[0025] Figure 4 is a block diagram of an exemplary software process illustrating the validation and execution of test classes within a test plan.

[0026] Figure 5 is a block diagram of an exemplary software process illustrating an implementation of the edit and update functionality in accordance with one embodiment of the present invention.

[0027] Figure 6 illustrates an on-screen graphical user interface configuration dialog box to let a user setup various settings for the edit and update functionality in accordance with one embodiment of the present invention.

[0028] Figure 7 is a block diagram illustrating the lifecycle of an edit and update session in accordance with one embodiment of the present invention.

[0029] Figure 8 is a flow diagram illustrating the invocation of the edit and update procedure within a flow item that is part of a branch flow within a concurrent flow item.

[0030] Figure 9 depicts a flowchart of an exemplary computer controlled process for implementing the edit and update functionality in accordance with one embodiment of the present invention.

[0031] DETAILED DESCRIPTION OF THE INVENTION

[0032] Reference will now be made in detail to the various embodiments of the present disclosure, examples of which are illustrated in the accompanying drawings. While described in conjunction with these embodiments, it will be understood that they are not intended to limit the disclosure to these embodiments. On the contrary, the disclosure is intended to cover alternatives, modifications and equivalents, which may be included within the spirit and scope of the disclosure as defined by the appended claims. Furthermore, in the following detailed description of the present disclosure, numerous specific details are set forth in order to provide a thorough understanding of the present disclosure. However, it will be understood that the present disclosure may be practiced without these specific details. In other instances, well-known methods, procedures, components, and circuits have not been described in detail so as not to unnecessarily obscure aspects of the present disclosure.

[0033] Some portions of the detailed descriptions that follow are presented in terms of procedures, logic blocks, processing, and other symbolic representations of operations on data bits within a computer memory. These descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. In the present application, a procedure, logic block, process, or the like, is conceived to be a self-consistent sequence of steps or instructions leading to a desired result. The steps are those utilizing physical manipulations of physical quantities. Usually, although not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a computer system. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as transactions, bits, values, elements, symbols, characters, samples, pixels, or the like.

[0034] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present disclosure, discussions utilizing terms such as "receiving," "saving," "suspending," "unloading," "compiling," "reloading," "resuming," "associating," "executing," "removing," "releasing," "reserving," "setting," "accessing," "freeing," "controlling," "adding," "determining," "identifying" or the like, refer to actions and

processes (e.g., flowchart 900 of Figure 9) of a computer system or similar electronic computing device or processor (e.g., system 110 of Figure 1). The computer system or similar electronic computing device manipulates and transforms data represented as physical (electronic) quantities within the computer system memories, registers or other such information storage, transmission or display devices.

[0035] Embodiments described herein may be discussed in the general context of computer-executable instructions residing on some form of computer-readable storage medium, such as program modules, executed by one or more computers or other devices. By way of example, and not limitation, computer-readable storage media may comprise non-transitory computer-readable storage media and communication media; non-transitory computer-readable media include all computer-readable media except for a transitory, propagating signal. Generally, program modules include routines, programs, objects, components, data structures, etc., that perform particular tasks or implement particular abstract data types. The functionality of the program modules may be combined or distributed as desired in various embodiments.

[0036] Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer-readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, random access memory (RAM), read only memory (ROM), electrically erasable programmable ROM (EEPROM), flash memory or other memory technology, compact disk ROM (CD-ROM), digital versatile disks (DVDs) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium that can be used to store the desired information and that can accessed to retrieve that information.

[0037] Communication media can embody computer-executable instructions, data structures, and program modules, and includes any information delivery media. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, radio frequency (RF), infrared, and other wireless media. Combinations of any of the above can also be included within the scope of computer-readable media.

[0038] Figure 1 is a block diagram of an example of a computing system 110 for a system controller capable of implementing embodiments of the present disclosure. For example, computing system 110, in one embodiment, could implement the controller of the tester system. Computing system 110 broadly represents any single or multi-processor computing device or system capable of executing computer-readable instructions. Examples of computing system 110 include, without limitation, workstations, laptops, client-side terminals, servers, distributed computing systems, or any other computing system or device. In its most basic configuration, computing system 110 may include at least one processor 114 and a system memory 116.

[0039] Processor 114 generally represents any type or form of processing unit capable of processing data or interpreting and executing instructions. In certain embodiments, processor 114 may receive instructions from a software application or module. These instructions may cause processor 114 to perform the functions of one or more of the example embodiments described and/or illustrated herein.

[0040] System memory 116 generally represents any type or form of volatile or non-volatile storage device or medium capable of storing data and/or other computer-readable instructions. Examples of system memory 116 include, without limitation, RAM, ROM, flash memory, or any other suitable memory device. Although not required, in certain embodiments computing system 110 may include both a volatile memory unit (such as, for example, system memory 116) and a non-volatile storage device (such as, for example, primary storage device 132).

[0041] Computing system 110 may also include one or more components or elements in addition to processor 114 and system memory 116. For example, in the embodiment of Figure 1, computing system 110 includes a memory controller 118, an input/output (I/O) controller 120, and a communication interface 122, each of which may be interconnected via a communication infrastructure 112. Communication infrastructure 112 generally represents any type or form of infrastructure capable of facilitating communication between one or more components of a computing device. Examples of communication infrastructure 112 include, without limitation, a communication bus (such as an Industry Standard Architecture (ISA), Peripheral Component Interconnect (PCI), PCI Express (PCIe), or similar bus) and a network.

[0042] Memory controller 118 generally represents any type or form of device capable of handling memory or data or controlling communication between one or more components of computing system 110. For example, memory controller 118 may control communication between processor 114, system memory 116, and I/O controller 120 via communication infrastructure 112.

[0043] I/O controller 120 generally represents any type or form of module capable of coordinating and/or controlling the input and output functions of a computing device. For example, I/O controller 120 may control or facilitate transfer of data between one or more elements of computing system 110, such as processor 114, system memory 116, communication interface 122, display adapter 126, input interface 130, and storage interface 134.

[0044] Communication interface 122 broadly represents any type or form of communication device or adapter capable of facilitating communication between example computing system 110 and one or more additional devices. For example, communication interface 122 may facilitate communication between computing system 110 and a private or public network including additional computing systems. Examples of communication interface 122 include, without limitation, a wired network interface (such as a network interface card), a wireless network interface (such as a wireless network interface card), a modem, and any other suitable interface. In one embodiment, communication interface 122 provides a direct connection to a remote server via a direct link to a network, such as the Internet. Communication interface 122 may also indirectly provide such a connection through any other suitable connection.

[0045] Communication interface 122 may also represent a host adapter configured to facilitate communication between computing system 110 and one or more additional network or storage devices via an external bus or communications channel. Examples of host adapters include, without limitation, Small Computer System Interface (SCSI) host adapters, Universal Serial Bus (USB) host adapters, IEEE (Institute of Electrical and Electronics Engineers) 1394 host adapters, Serial Advanced Technology Attachment (SATA) and External SATA (eSATA) host adapters, Advanced Technology Attachment (ATA) and Parallel ATA (PATA) host adapters, Fibre Channel interface adapters, Ethernet adapters, or the like. Communication interface 122 may also allow computing system 110 to engage in distributed or remote

computing. For example, communication interface 122 may receive instructions from a remote device or send instructions to a remote device for execution.

[0046] As illustrated in Figure 1, computing system 110 may also include at least one display device 124 coupled to communication infrastructure 112 via a display adapter 126. Display device 124 generally represents any type or form of device capable of visually displaying information forwarded by display adapter 126. Similarly, display adapter 126 generally represents any type or form of device configured to forward graphics, text, and other data for display on display device 124.

[0047] As illustrated in Figure 1, computing system 110 may also include at least one input device 128 coupled to communication infrastructure 112 via an input interface 130. Input device 128 generally represents any type or form of input device capable of providing input, either computer- or human-generated, to computing system 110. Examples of input device 128 include, without limitation, a keyboard, a pointing device, a speech recognition device, or any other input device.

[0048] As illustrated in Figure 1, computing system 110 may also include a primary storage device 132 and a backup storage device 133 coupled to communication infrastructure 112 via a storage interface 134. Storage devices 132 and 133 generally represent any type or form of storage device or medium capable of storing data and/or other computer-readable instructions. For example, storage devices 132 and 133 may be a magnetic disk drive (e.g., a so-called hard drive), a floppy disk drive, a magnetic tape drive, an optical disk drive, a flash drive, or the like. Storage interface 134 generally represents any type or form of interface or device for transferring data between storage devices 132 and 133 and other components of computing system 110.

[0049] In one example, databases 140 may be stored in primary storage device 132. Databases 140 may represent portions of a single database or computing device or it may represent multiple databases or computing devices.

[0050] Continuing with reference to Figure 1, storage devices 132 and 133 may be configured to read from and/or write to a removable storage unit configured to store computer software, data, or other computer-readable information. Examples of suitable removable storage

units include, without limitation, a floppy disk, a magnetic tape, an optical disk, a flash memory device, or the like. Storage devices 132 and 133 may also include other similar structures or devices for allowing computer software, data, or other computer-readable instructions to be loaded into computing system 110. For example, storage devices 132 and 133 may be configured to read and write software, data, or other computer-readable information. Storage devices 132 and 133 may also be a part of computing system 110 or may be separate devices accessed through other interface systems.

[0051] Many other devices or subsystems may be connected to computing system 110. Conversely, all of the components and devices illustrated in Figure 1 need not be present to practice the embodiments described herein. The devices and subsystems referenced above may also be interconnected in different ways from that shown in Figure 1. Computing system 110 may also employ any number of software, firmware, and/or hardware configurations. For example, the example embodiments disclosed herein may be encoded as a computer program (also referred to as computer software, software applications, computer-readable instructions, or computer control logic) on a computer-readable medium.

[0052] The computer-readable medium containing the computer program may be loaded into computing system 110. All or a portion of the computer program stored on the computer-readable medium may then be stored in system memory 116 and/or various portions of storage devices 132 and 133. When executed by processor 114, a computer program loaded into computing system 110 may cause processor 114 to perform and/or be a means for performing the functions of the example embodiments described and/or illustrated herein. Additionally or alternatively, the example embodiments described and/or illustrated herein may be implemented in firmware and/or hardware.

[0053] For example, a computer program for running test plans may be stored on the computer-readable medium and then stored in system memory 116 and/or various portions of storage devices 132 and 133. When executed by the processor 114, the computer program may cause the processor 114 to perform and/or be a means for performing the functions required for sharing resources between multiple test cores in a concurrent test environment.

IMPLEMENTING EDIT AND UPDATE FUNCTIONALITY WITHIN A DEVELOPMENT ENVIRONMENT USED TO COMPILE TEST PLANS FOR AUTOMATED SEMICONDUCTOR DEVICE TESTING

[0054] The conventional edit and continue functionality found in some development environments has various limitations. First, it cannot be used in connection with all types of code. For example, it cannot be used in a debugging environment when debugging 64-bit code. As a result, many users of tester software that develop 64-bit test classes and test plans for performance reasons are unable to take advantage of this feature.

[0055] Second, there are limitations on the type of code changes allowed. For example, one drawback is that conventional development environments only allow relatively minor code changes to be made when using the feature. Typically, the altered code section is recompiled, and a small portion of executing memory corresponding to the altered code is swapped out and replaced with a new block of memory generated using the new code. Thereafter, the appropriate instruction pointers are set accordingly.

[0056] The feature, as designed in conventional systems, is not able to handle any intricate code alterations. As a result, it is not very useful for developers of test classes for ATE. If users make a substantial error in the preparation of the test class or need to replace a test class for any other reason while a test plan is executing, they typically have to stop the test plan and unload it first. Then, they would need to reload the test plan and restart it from the beginning before getting back to the original point in the code. Such a work flow is time-consuming and error-prone.

[0057] Accordingly, the embodiments of the present invention provide an automated method and system to allow users to make changes to test classes during execution of the test plan without needing to manually perform all the above discussed steps. This allows developers to prepare and debug test classes in a significantly more efficient and less error-prone manner than they were allowed to using conventional systems.

[0058] In order to accomplish this objective, embodiments of the present invention provide edit and update functionality within tester software systems. While debugging, a user may set a

breakpoint in the test class source code causing the system to stop. The user can edit the source code, thereby initiating an edit and update session. The state information is then stored and the edited test classes are unloaded. Subsequently, the edit and update procedure will recompile the edited version of the test class and swap the original version with the newly compiled version. The edited test class is then re-executed and automatically stepped back to the same breakpoint as where the user left off debugging and the previously stored state information is restored. The test plan can then continue with the newly edited code.

[0059] Using embodiments of the present invention, developers can make substantial changes to the test classes and are not limited to only implementing minor edits. In fact, one embodiment of the present invention supports changing the entire test class file instead of only a minor section of the file.

[0060] Figure 2A is a schematic block diagram for an automated test equipment (ATE) apparatus on which embodiments of the edit and update procedure can be implemented in accordance with one embodiment of the present invention. In one embodiment, the system controller 201 comprises one or more linked computers. For example, testing systems such as Advantest Corporation's T2000 tester family, can use a network of computers. In other embodiments, the system controller often comprises only a single computer. The system controller 201 is the overall system control unit, and runs the software for the ATE that is responsible for accomplishing all the user-level testing tasks, including running the user's main test plan. One example of such tester software is the T2000 System Software. Typically, tester software such as the T2000 comprises a compiler, which compiles test plans written in the programming language of the tester system, e.g. OPENSTAR Test Programming Language ("OTPL").

[0061] The communicator bus 215 provides a high-speed electronic communication channel between the system controller and the tester hardware. The communicator bus can also be referred to as a backplane, a module connection enabler, or system bus. Physically, communicator bus 215 is a fast, high-bandwidth duplex connection bus that can be electrical, optical, etc. In one embodiment, communicator bus 215 can use the TCP/IP protocol. System controller 201 sets up the conditions for testing the DUTs 211-214 by programming the tester hardware through commands sent over the communicator bus 215.

[0062] Tester hardware 202 comprises the complex set of electronic and electrical parts and connectors necessary to provide the test stimulus to the devices under test (DUTs) 211-214 and measure the response of the DUTs to the stimulus, and compare it against the expected response.

[0063] Figure 2B is a more detailed schematic block diagram of one embodiment of the automated test equipment apparatus of Figure 2A. In the embodiment illustrated in Figure 2B, tester hardware 202 can comprise multiple site controllers 270, wherein each site controller is connected to multiple DUTs. Each site controller is a computer used in a device test. A test plan program can be executed on a site controller, which controls test modules 280 in the course of performing device tests for DUTs 290. The site controllers 270 are connected to the system controller, which controls the site controllers over communicator bus 215. In some embodiments, test developers cannot directly operate the site controllers but instead operations performed by the developer are processed on the system controller 201, which controls the site controllers via communicator bus 215.

[0064] Site controllers 270 and test modules 280, in one embodiment, can be connected via high-speed optical buses. The bus switch 285 has a matrix structure for connecting the site controllers with the test modules. Using the bus switch 285 allows a site controller to connect with any test modules 280, and allows flexibility in configuring bus connections.

[0065] Test modules 280 required for device test are typically mounted in the test system test head. Test module configuration can be adapted to the targeted device. A set of test modules 280 required for device test is called a test site. Each test site 295 is controlled by a site controller. There can be various types of test modules for many applications. Some exemplary types of modules are sync-generator module, sync-matrix module, device power supply module, analog module, RF module and digital module.

[0066] Figure 3 illustrates a diagram regarding the manner in which different test instances can be created from a single test class. As discussed in detail above, test classes allow the user to configure class behavior by providing parameters that are used to specify the options for a particular instance of that test. For example, a test class for implementing a functional test may provide parameters to specify a test pattern to execute on the DUTs being tested. Also it may

provide parameters to specify the level and timing conditions for the test. Specifying different values for these parameters allows the user to create different instances of the functional test.

[0067] In a tester software system such as T2000, a test plan can be described in a test programming language such as OTPL, as discussed above. OTPL can be used to describe test execution sequence and the corresponding test conditions used. While OTPL can describe a test plan, it typically does not describe any test algorithm. The test algorithm in a system such as T2000, for example, can be described using the C++ language in a test class as discussed above.

[0068] While the discussion of the invention is in the context of the T2000 tester software, wherein test plans are developed using OTPL and test classes are developed using C++, the present invention is not limited to this embodiment. Other types of tester software using different programming languages may be employed in other embodiments. It will be understood to one of ordinary skill in the art that the teachings of the present invention can cover alternatives, modifications and equivalents, which may be included within the spirit and scope of the disclosure as defined by the appended claims.

[0069] In the example from Figure 3, test class 310 can be used to create three separate instances of the test "TestTypeX." Each of the three instances, Instance1 320, Instance2 340, and Instance3 350, receives a separate set of parameter data for each instance of the test. For example, Instance1 320 receives parameter data 330, while the remaining instances receive their own set of parameters.

[0070] Figure 4 is a schematic block diagram of an exemplary software process illustrating the validation and execution of test classes within a test plan.

[0071] In one embodiment, the user enters in all the information regarding the requisite parameters for the various test classes and other general properties into the pre-header file 410. The information regarding the parameters may comprise the names of the parameters, their allowed values, their types etc.

[0072] In conventional tester systems, the tester system software can provide a mechanism for verifying that the appropriate parameters are available for inclusion in the generated source code. The tester system software provides a method that allows the test class developer to fully

specify, in a text-based source file per test class, the public methods and attributes of the test class that the developer has designated are the ones required to parameterize the class. Specifying its methods and attributes in a text file allows the tester software to easily determine if the appropriate parameters are available, which facilitates the error checking and validation of the test plan during the translation phase. In conventional ATE systems, this text-based description is embedded in a pre-header file for the test class, which is used by the compiler in the tester software to determine the methods and attributes of the test class. Further, it is used by the compiler for generating the header for the test class. In one embodiment, the header for the test class, similar to the test class itself, may also be in a conventional programming language such as C++. The generated C++ header file is the one used to finally compile the test class C++ code.

[0073] As discussed, the pre-header file is used by the tester system software to automatically generate the test class declaration C++ header file 415 from the pre-header file directly. The generated C++ header file is the one used to finally compile the test class C++ code 435. The C++ test class code 435 is compiled into binary DLL files that can be loaded for execution in the tester system software 420.

[0074] During validation phase, the tester system software 420 (also referred to as "tester operating system") reads in and analyzes the test plan OTPL code 450 developed by the test plan author for errors. The pre-header file 410 is used by tester operating system 420 to describe the test classes and the parameters needed by the test classes. By describing the test classes, the pre-header file allows the tester system software 420 to validate that the proper parameters are being passed to the various instances of the test classes.

[0075] The OTPL code 450 then gets loaded into the tester system software 420 along with the binary DLL test class files 475. In one embodiment, after OTPL code 450 is validated, the test plan code 450 then instantiates the test classes and uses the information from the pre-header file 410 to determine the appropriate parameters for the test class instantiation. The tester operating system 420 can populate the parameters for the various test classes based on the information provided in the pre-header file 410. For example, in Figure 4, the tester operating system may read in a pattern list 490 and a pattern 495 that may be used as parameters for certain

test classes instantiated within the OTPL code 450. The test plan 425 is then executed by the tester system software 420.

[0076] Figure 5 is a block diagram of an exemplary software process illustrating an implementation of the edit and update functionality in accordance with one embodiment of the present invention.

[0077] In one embodiment, at block 501, the user launches the develop environment, e.g. Visual Studio and loads a test class project that the user wants to debug using the application. At block 502, the user starts the tester system software ("TSS"). After TSS is started, the user loads a test plan or test program at 503. The user will then attach the debugger to the site controller, which may be connected to a DUT being tested, at block 504 to start the debugging session. The user also sets a breakpoint in a method of the test class that they want to debug. For example, the user could set a breakpoint in a method called "preExec()", which is one of the methods comprising FlowItem2 519 as shown in block 508. Other exemplary methods comprising FlowItem 519 are "execute()" and "postExec()" at blocks 509 and 510 respectively.

[0078] The user will then execute the testflow at block 505. The first flow item, FlowItem1, at block 518 is executed first. However, during execution of the FlowItem2 at block 519, the debugger will break at the breakpoint in preExec() at block 508, and the source code may be visible in the code window of the debugging environment. At this point, the user may single step through the code until the user reaches a point where edits need to be made to the code. The user can make edits to the code in the same file or other files associated with the test class project. In one embodiment, instead of making edits, the user may also simply leave the debugger and continue execution.

[0079] If edits are made, then after completing the edits, the user may invoke the edit and update procedure at block 516. For example, the user may invoke the edit and update at method "AAA()" inside preExec() when executing Flow item2. In one embodiment, when the edit and update procedure is invoked, the execution of the entire test class method being debugged, i.e., methods preExec(), execute() and postExec() will be completed before the tester operating system is suspended at block 511. In another embodiment, the edit and update procedure may suspend right away without finishing execution of the remaining methods in the test flow item.

In this embodiment, the user can immediately skip out of executing any remaining test class code and then suspend execution of the test plan to allow rebuilding and reloading of the test class.

[0080] In one embodiment, the edit and update procedure will keep track of the line at which it was invoked, so that it can return the user to the same line (hereinafter referred to as the "current line") after the test class has been updated and reloaded.

[0081] At block 512, the debugger is detached from the site controller it attached to in block 504. The site controllers 270 are typically where the test classes are executing. The system controller 201 is typically where the development environment runs. Accordingly, the user is typically implementing a remote debugging session from the system controller 201 on the site controller 270.

[0082] The debugger detaches from the site controller at block 512 and the effected test classes are unloaded. In the example of Figure 5, any test class associated with FlowItem2 519 and any other class that inherits from it will need to be unloaded. This is because if the procedure is to alter memory with respect to an effected "base" class, it also needs to alter memory with respect to other "derived" classes that depend from the base test class. Accordingly, the edit and update procedure first needs to identify all the derived test classes. Next, the base test class and all derived test classes that depend on the base test class will be unloaded. Because the test classes are DLLs at this stage, they need to be unloaded before they can be edited and updated. Without unloading the effected test class DLLs, the test plan executable, which is in its suspended state, still has a handle on them, which precludes them from being written to.

[0083] At block 513, the edit and update procedure saves the source test class files modified in the code window of the development environment and then compiles the test class project to generate a new DLL. The new DLL is transmitted down from system controller 201 to site controller 270. In addition to the DLL for the base test class that was modified, DLLs for the derived test classes will also need to be generated and transmitted to the site controller 270.

[0084] If the test class project compiles successfully, the updated test class DLL will be reloaded back at step 514. Alternatively, in one embodiment, instead of generating an updated

test class DLL from the edited test class file, a new version of the test class file may be created and compiled on the system controller 201 and, subsequently, transmitted down to the site controller 270, so it can be reloaded back into the TSS. Reloading the test class DLL comprises populating the test instances of the test class being debugged with the same test parameter values that were set originally using the new test class DLL. In addition, it also means populating the instances of all the derived test classes as well.

[0085] At block 515, the debugger is attached back to the same site controller that was being debugged in the prior debug session. The edit and update procedure, in one embodiment, will set a breakpoint at the "current line" at which the instruction pointer was paused when the edit and update procedure was initiated. In the example of Figure 5, the breakpoint would be set at method AAA(). Other breakpoints that were used in the previous debug session will be disabled temporarily so that the user can be returned to the current line in the code where the edit and update process was invoked instead of at an earlier breakpoint that may have been set. In different embodiments, other ways to control the breakpoints may be provided as the current line may have been removed while editing the test class. For example, the user could be returned to the closest line before the current line within the test class code. Or the user could be returned to the beginning of the function containing the current line.

[0086] Subsequently, the flow item, Flow item2, associated with the updated test class is re-executed by TSS and the execution will break within the debugger at the current line, i.e., at method AAA(). The edit and update procedure will then restore all the temporarily disabled breakpoints that had been set by the user. Thereafter, the edit and update process ends at block 517. An edit and update session can typically be defined to last from the point of time the user selects the edit and update procedure to execute till the point of time when the debugged flow item is about to be re-executed from the point or close to the point at which the user left off when edit and update was invoked.

[0087] After the edit and update process ends, TSS resumes execution of the flow if the user chooses to continue execution. Accordingly, the user can choose to finish executing Flow item2 at block 519, and Flow item3 at block 520.

[0088] After the flow is finished executing, the user can choose to stop TSS at block 506 and shut down the development environment at block 507.

[0089] As discussed above, one of the advantages of the edit and update functionality is to allow users of the tester software system to edit their test class code and re-run their tests in the debugging environment without requiring the entire test plan to be reloaded and without burdening the user with manually going through a series of steps to accomplish test class modifications. Further, because the test class DLLs are unloaded, unlike the conventional "edit and continue" procedure, the user has the flexibility of making much more significant changes to the code of the test class, and in some cases even replacing the entire test class code.

[0090] Further, by allowing suspension of the test plan, the test classes can be unloaded and reloaded and the user can be returned to resume debugging the code at the same line in the code where the user had paused when she had started making edits and invoked the edit and update procedure. By comparison, a user who tried to update the test classes without using the edit and update procedure would need to cancel execution of the test plan completely to edit the test class. When execution of the test plan is cancelled, the user is returned to a ready state, which entails starting execution from the beginning of the test plan. Executing from the beginning of the test plan would be relatively cumbersome, especially in instances where the flow item the user updated was towards the end of the test plan. The edit and update procedure, on the other hand, allows the user to simply start at the beginning of the test class that was edited and return to the current line within the test flow item that the user was executing when the edit and update procedure was invoked.

[0091] Moreover, using the edit and update functionality advantageously ensures that the tester state is maintained, because the test plan is merely being suspended as opposed to cancelled. If a test plan is cancelled and execution needs to start from the beginning, the tester state may not be maintained if the system is not deterministic. For example, if a user is debugging a device, small changes in the device state during testing, e.g., heating up or cooling down of the device could cause the test to produce different results. Thus, a user may never actually get back to a particular flow item depending on how a DUT behaves during a test. Using the edit and update functionality avoids this problems because the user's location within the test plan never changes, the test plan merely suspends and the user is returned to the same

location as before in the test plan after the test classes are reloaded back. As discussed above, the TSS provides users the ability to make modifications to test class code in the middle of a debugging session and continue testing the device without unloading the test plan.

[0092] Figure 6 illustrates an on-screen graphical user interface configuration dialog box to let a user setup various settings for the edit and update functionality in accordance with one embodiment of the present invention. The radio button group with label Code Update Mode 680 in the configuration dialog box as shown in Figure 6 can provide three radio buttons to let the user choose between edit and update 605, conventional "edit and continue" 615 or none 640. The none 640 option allows the user to keep debugging edited code when an update is made. Of course, as mentioned above, in circumstances where "edit and continue" cannot be used due to its various limitations, e.g., 64 bit environments, the "edit and continue" radio button will be disabled and thus cannot be selected in such configurations.

[0093] In one embodiment, edit and update can be invoked by debug commands by selecting option 610 in the configuration dialog box. This means that edit and update will be automatically invoked whenever the user executes the test class in the development environment using a debug function, e.g., step execution, continuous execution, etc. Stated differently, if there is any code change related to the current test class under debug or its derivative test classes, edit and update will be automatically invoked when the user uses any debug commands to continue test execution.

[0094] In one embodiment, the user can also be allowed to select the breakpoint entry mode 690. The breakpoint entry mode allows the user to select the point at which control of debugging is to be returned to the user. If the user selects the current line option 615, a breakpoint to the current line will be added once the debugger is paused and all other breakpoints are temporarily disabled. When the flow item is re-executed, this breakpoint will be hit and control will be given back to the user after re-enabling all the temporarily disabled breakpoints.

[0095] If the user selects the beginning of current function option 620, instead of current line, a breakpoint will be added to the opening brace of the function which contains the current line. After the breakpoint is hit at this breakpoint, all other breakpoints that were temporarily disabled at the start of the edit and update session will be enabled.

[0096] If the user selects the any previous breakpoints option 625, after the test class is reloaded, all breakpoints that the user set in the previous debug sessions at which edit and update was invoked will be enabled. Also, no additional breakpoints will be added. When the flow item is restarted, since all breakpoints are enabled, if any line containing a breakpoint is executed, execution of the flow item will be paused there.

[0097] As discussed above, an edit and update session can typically be defined to last from the point of time the user selects the edit and update procedure to execute till the point of time when the debugged flow item is about to be re-executed. Figure 7 is a block diagram illustrating the lifecycle of an edit and update session in accordance with one embodiment of the present invention. As illustrated in Figure 7, an edit and update session may be overlapped by more than one debug session.

[0098] In Figure 7, the edit and update session 725 is overlapped by two debug sessions, 710 and 745. The test class DLLs are compiled and reloaded between the two debug sessions, 710 and 745. The edit and update session 725 is invoked at step 715 after debug session 710 starts at step 705. The debug session 710 exits at step 720 and detaches from the site controller at step 780. After the debugger detaches, the modified base class and its derivative classes are compiled and reloaded at 730. Following reload, debug session 745 attaches to the site controller at step 785 and the modified base class is re-run at 735 till the breakpoint selected by the user is reached. At that point, the edit and update process is complete at 740 and TSS can resume execution of the test flow. It should be noted that typically the user is allowed to invoke only one edit and update session at any time.

[0099] In one embodiment, the test class author is allowed to use the edit and update procedure within flow items which are part of a branch flow within a concurrent flow item. When debugging test class code in configurations where multiple flow items are executing in parallel, all flow executions except the one being debugged will be completed and the next scheduled flow item execution(s) will not be started until the edit and update session has finished. Stated differently, all flow executions that were running in parallel when the debugger hit a breakpoint will be completed first, but no other scheduled flow items will start executing until the edit and update process has finished. Using edit and update for a test class being

executed in a concurrent branch will result in the reloading of modified DLLs only after the other branches complete the execution of the entire current flow item.

[00100] Figure 8 is a flow diagram illustrating the invocation of the edit and update procedure within a flow item that is part of a branch flow within a concurrent flow item. Step 1 in Figure 8 illustrates a point in time when the user has stopped at a breakpoint in the test class associated with flow item Y 802 while P 801 and A 803 are also paused. The TSS will suspend the flow execution of the Y 802-Z 804 branch being debugged, i.e., the branch in which the breakpoint was encountered, and resume the other concurrent threads which were also paused by the debugger. The other branches (i.e., the P 801 branch and the A 803-B 805 branch) will be allowed to complete the currently executing flow item.

[00101] Step 2 in Figure 8 illustrates the point where P 801 and A 803 have completed execution as a result of the edit and update operation being initiated while Y 802 has not yet completed but is suspended to allow its test class DLL(s) to be rebuilt and reloaded.

[00102] Step 3 in Figure 8 illustrates the point after the edit and update procedure has completed and Y 802 is re-executed while flow item B 805 of the other branch starts executing. In one embodiment, the flow items in the other concurrent branches e.g. P 801 and A 803, are not retested after the unload-compile-reload test class cycle as they were not in the active stack being debugged while the user initiated the edit and update procedure. When the TSS starts to re-execute the flow item which invoked the "edit and update," the TSS will continue execution of these other branches from their respective suspend points.

[00103] In one embodiment, the edit and update procedure is configured to handle all error conditions in a safe manner so as to preserve the state of the tester. For example, if the edit and update process is interrupted for any reason, e.g. a compilation error raised when compiling edited test class code, the user might want to continue the edit and update session voluntarily after fixing the compilation errors. Alternatively, the user might want to abort the session if the user realizes that a fix requires the test plan file to be modified and thus requires a reloading of the test plan. Thus, the user is provided the capability to start an edit and update session and either continue or abort an ongoing "edit and continue" session.

[00104] If a user decides to abort the edit and update operation and reload the entire test plan, or if the TSS aborts an ongoing session in case an alarm or an unexpected exception is raised, the system state will depend on the time when aborting takes place. For example, if aborting takes place prior to entering suspend state, i.e. where TSS has resumed test plan execution but before all the flow execution branches get suspended, test plan execution will be cancelled and also the edit and update operation will be cancelled. The debugger will still be attached to the site controller in this instance and the older test class DLL will still be loaded. The previously disabled breakpoints will be re-enabled, and in order for the user to debug this test class, the user will need to re-execute the flow item. Similarly, in other circumstances, depending on when the edit and update operation is aborted, the system software is configured to perform all the necessary bookkeeping steps required to maintain a consistent tester state.

[00105] Typically, a flow item is started with a set of active DUTs. In one embodiment, the user may either put some DUTs in a hold state or reject some DUTs explicitly in the test class code before invoking edit and update. When the TSS starts to re-execute the flow item, the TSS will put the items on hold back to a measurable DUT state again. Accordingly, when re-executing the flow item, except DUTs that were explicitly rejected by the test class code, the same set of DUTs that were used in the previous execution will be used.

[00106] Figure 9 depicts a flowchart 900 of an exemplary computer controlled process for implementing the edit and update functionality in accordance with one embodiment of the present invention.

[00107] At step 901, the user edits the test class code and invokes the edit and update procedure.

[00108] At step 902, the procedure identifies the base class and all derived test classes that depend from the base test class as discussed in connection with Figure 5 above. Subsequently, at step 902, the test plan is suspended and the state information is stored. For example, in one embodiment, the state information comprising the line at which the user invoked the edit and update procedure is stored. In one embodiment, the test plan may suspend after finishing execution of the remaining test class from which the edit and update functionality was invoked.

In other embodiments, the user can immediately skip out of executing any remaining test class code and suspend execution of the test plan.

[00109] At step 904, the debugger detaches from the site controller 270 and the effected base and derived test classes are unloaded.

[00110] At step 906, the edit and update procedure saves the source test class files modified in the code window of the development environment and then compiles the test class project to generate one or more new DLLs. The new DLL(s) are transmitted down from system controller 201 to site controller 270. In addition to the DLL for the base test class that was modified, DLLs for the derived test classes will also need to be generated and transmitted to the site controller 270.

[00111] If the test class project compiles successfully, the updated test class DLL will be reloaded back at step 908 and the debugger is attached back to the same site controller that was being debugged in the prior debug session. Reloading the test class DLL comprises populating the test instances of the test class being debugged using the new test class DLL and with the same test parameter values that were set originally. In addition, it also means populating the instances of all the derived test classes as well.

[00112] At step 910, the edit and update procedure, in one embodiment, disables temporarily all the breakpoints that were used in the previous debug session so the user can be returned to the current line in the code or in the vicinity of the code where the edit and update process was invoked. At step 912, the procedure will set a breakpoint at the "current line" at which the instruction pointer was paused when the edit and update procedure was initiated. In different embodiments, other ways to control the breakpoint that the user is returned to may be provided as detailed above.

[00113] Subsequently, the flow item associated with the updated test class is re-executed by TSS at step 914 from the beginning of the test class instance. At step 916, the execution will break within the debugger at the current line or other location that the user previously elected to return to through the breakpoint entry mode choice 690. At step 918, the edit and update

procedure will then restore all the temporarily disabled breakpoints that had been set by the user. Thereafter, the edit and update process ends.

[00114] While the foregoing disclosure sets forth various embodiments using specific block diagrams, flowcharts, and examples, each block diagram component, flowchart step, operation, and/or component described and/or illustrated herein may be implemented, individually and/or collectively, using a wide range of hardware, software, or firmware (or any combination thereof) configurations. In addition, any disclosure of components contained within other components should be considered as examples because many other architectures can be implemented to achieve the same functionality.

[00115] The process parameters and sequence of steps described and/or illustrated herein are given by way of example only. For example, while the steps illustrated and/or described herein may be shown or discussed in a particular order, these steps do not necessarily need to be performed in the order illustrated or discussed. The various example methods described and/or illustrated herein may also omit one or more of the steps described or illustrated herein or include additional steps in addition to those disclosed.

[00116] While various embodiments have been described and/or illustrated herein in the context of fully functional computing systems, one or more of these example embodiments may be distributed as a program product in a variety of forms, regardless of the particular type of computer-readable media used to actually carry out the distribution. The embodiments disclosed herein may also be implemented using software modules that perform certain tasks. These software modules may include script, batch, or other executable files that may be stored on a computer-readable storage medium or in a computing system. These software modules may configure a computing system to perform one or more of the example embodiments disclosed herein. One or more of the software modules disclosed herein may be implemented in a cloud computing environment. Cloud computing environments may provide various services and applications via the Internet. These cloud-based services (e.g., software as a service, platform as a service, infrastructure as a service, etc.) may be accessible through a Web browser or other remote interface. Various functions described herein may be provided through a remote desktop environment or any other cloud-based computing environment.

[00117] The foregoing description, for purpose of explanation, has been described with reference to specific embodiments. However, the illustrative discussions above are not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations are possible in view of the above teachings. The embodiments were chosen and described in order to best explain the principles of the invention and its practical applications, to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as may be suited to the particular use contemplated.

[00118] Embodiments according to the invention are thus described. While the present disclosure has been described in particular embodiments, it should be appreciated that the invention should not be construed as limited by such embodiments, but rather construed according to the below claims.

[0001] CLAIMS

What is claimed is:

1. A method for debugging test procedures for automated device testing, said method comprising:

receiving a command to update at least one modified test procedure modified during a first debugging session;

saving state information for a test plan, wherein said state information comprises information regarding a breakpoint entry location, and wherein said modified test procedure is invoked within said test plan;

suspending execution of said test plan;

unloading said modified test procedure from said test plan;

compiling said modified test procedure to produce a compiled file associated with said modified test procedure;

reloading said modified test procedure into said test plan using said compiled file;

resuming execution of said modified test procedure in a second debugging session; and

breaking said execution during said second debugging session at a breakpoint corresponding to said breakpoint entry location.

2. The method of Claim 1, wherein said resuming further comprises:

clearing prior breakpoints set during said first debugging session;

setting a breakpoint in said modified test procedure corresponding to said breakpoint entry location; and

restoring said prior breakpoints after execution breaks during said second debugging session at said breakpoint corresponding to said breakpoint entry location.

3. The method of Claim 1, wherein said reloading further comprises:

transferring said compiled file from a system controller to a site controller, wherein said system controller is communicatively coupled to said site controller, and wherein said system controller controls said site controller;

instantiating test instances associated with said modified test procedure in said test plan using said compiled file; and

populating said test instances in said test plan with parameter values originally set during said first debugging session.

4. The method of Claim 1, wherein said modified test procedure comprises a base test procedure and at least one derived test procedure that depends from said base test procedure.

5. The method of Claim 4, further comprising:

identifying said at least one derived test procedure that depends from said base test procedure before said unloading.

6. The method of Claim 1, further comprising:

executing said modified test procedure to completion prior to said suspending.

7. The method of Claim 1, wherein said breakpoint entry location is selected from the group comprising: a current line, beginning of a current function, and any prior breakpoints.

8. The method of Claim 1, further comprising:

executing said test plan to completion from said breakpoint corresponding to said breakpoint entry location during said second debugging session.

9. The method of Claim 1, further comprising:

invoking said command to update said modified test procedure automatically when said modified test procedure is modified and a debug command is executed.

10. The method of Claim 1, further comprising:

determining if test procedures are running in parallel to said modified test procedure in concurrently executing branch flow items;

executing said test procedures running in parallel to completion while said modified test procedure is in a suspended state following said suspending; and

suspending execution of further test procedures in said concurrently executing branch flow items while said modified test procedure is executing.

11. A computer-readable storage medium having stored thereon, computer executable instructions that, if executed by a computer system cause the computer system to perform a method for debugging test procedures for automated device testing, said method comprising:

receiving a command to update at least one modified test procedure modified during a first debugging session;

saving state information for a test plan, wherein said state information comprises information regarding a breakpoint entry location, and wherein said modified test procedure is invoked within said test plan;

suspending execution of said test plan;

unloading said modified test procedure from said test plan;

compiling said modified test procedure to produce a compiled file associated with said modified test procedure;

reloading said modified test procedure into said test plan using said compiled file;

resuming execution of said modified test procedure in a second debugging session; and

breaking said execution during said second debugging session at a breakpoint corresponding to said breakpoint entry location.

12. The computer-readable medium as described in Claim 11, wherein said method further comprises:

clearing prior breakpoints set during said first debugging session;

setting a breakpoint in said modified test procedure corresponding to said breakpoint entry location; and

restoring said prior breakpoints after execution breaks during said second debugging session at said breakpoint corresponding to said breakpoint entry location.

13. The computer-readable medium as described in Claim 11, wherein said method further comprises:

transferring said compiled file from a system controller to a site controller, wherein said system controller is communicatively coupled to said site controller, and wherein said system controller controls said site controller;

instantiating test instances associated with said modified test procedure in said test plan using said compiled file; and

populating said test instances in said test plan with parameter values originally set during said first debugging session.

14. The computer-readable medium as described in Claim 11, wherein said modified test procedure comprises a base test procedure and at least one derived test procedure that depends from said base test procedure.

15. The computer-readable medium as described in Claim 14, wherein said method further comprises:

identifying said at least one derived test procedure that depends from said base test procedure before said unloading.

16. The computer-readable medium as described in Claim 11, wherein said method further comprises:

executing said modified test procedure to completion prior to said suspending.

17. The computer-readable medium as described in Claim 11, wherein said breakpoint entry location is selected from the group comprising: a current line, beginning of a current function, and any prior breakpoints.

18. The computer-readable medium as described in Claim 11, wherein said method further comprises:

executing said test plan to completion from said breakpoint corresponding to said breakpoint entry location during said second debugging session.

19. The computer-readable medium as described in Claim 11, wherein said method further comprises:

invoking said command to update said modified test procedure automatically when said modified test procedure is modified and a debug command is executed.

20. The computer-readable medium as described in Claim 11, wherein said method further comprises:

determining if test procedures are running in parallel to said modified test procedure in concurrently executing branch flow items;

executing said test procedures running in parallel to completion while said modified test procedure is in a suspended state following said suspending; and

suspending execution of further test procedures in said concurrently executing branch flow items while said modified test procedure is executing.

21. A system for debugging test procedures for automated device testing, said system comprising:

a memory comprising a development environment stored therein, wherein said development environment is operable to debug a test program, and wherein said development environment comprises a debugger;

a processor coupled to the memory, the processor being configured to operate in accordance with the development environment to:

save state information for a test plan, wherein said state information comprises information regarding a breakpoint entry location, and wherein a modified test procedure is invoked within said test plan;

suspend execution of said test plan;

unload said modified test procedure from said test plan;

compile said modified test procedure, wherein said compile generates a compiled file associated with said modified test procedure;

reload said modified test procedure into said test plan using said compiled file;

resume execution of said modified test procedure in a debugging session; and

breaking said execution during said debugging session at a breakpoint corresponding to said breakpoint entry location.

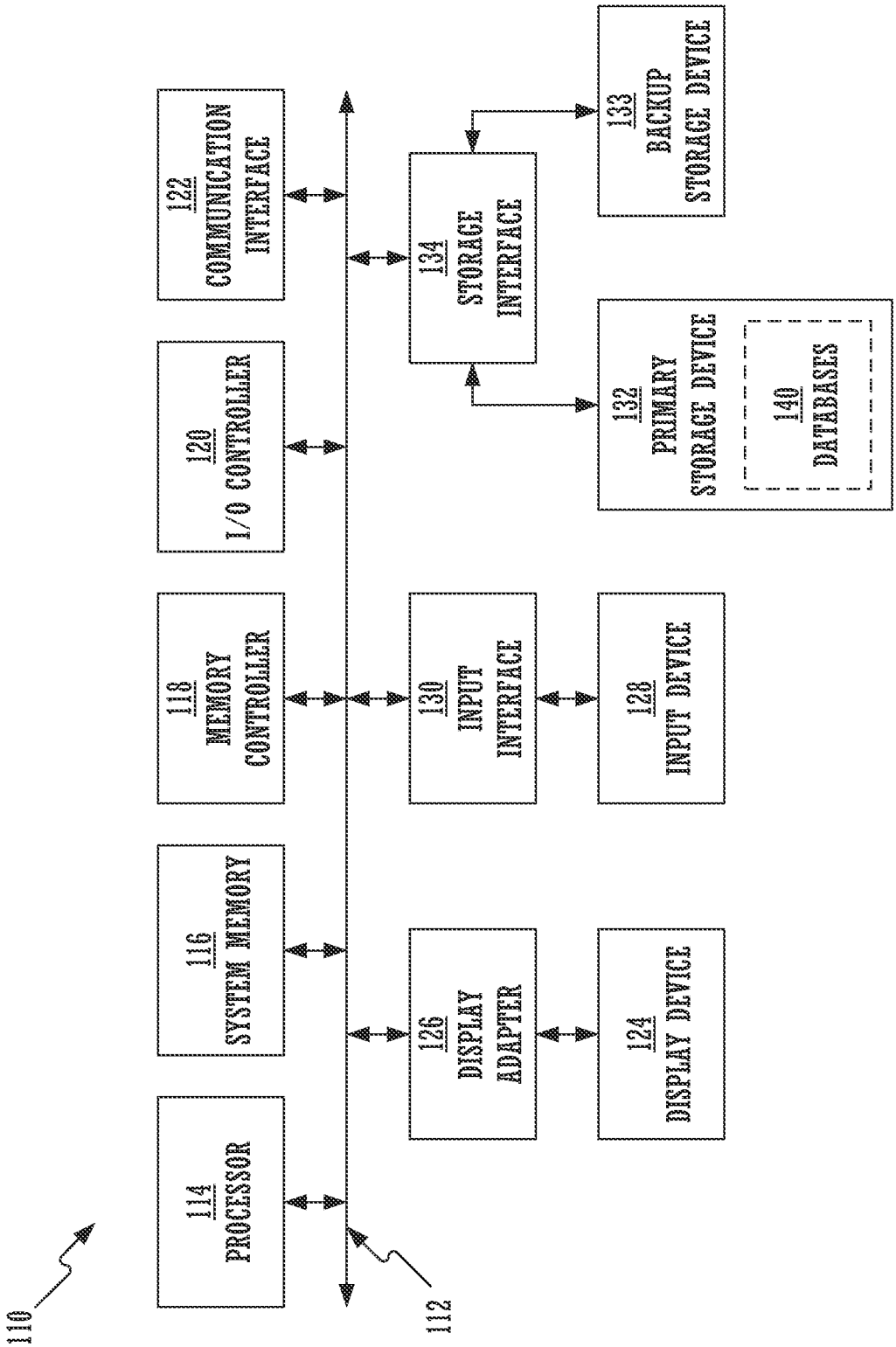
110
112
114 PROCESSOR
116 SYSTEM MEMORY
118 MEMORY CONTROLLER
120 I/O CONTROLLER
122 COMMUNICATION INTERFACE
124 DISPLAY DEVICE
126 DISPLAY ADAPTER
128 INPUT DEVICE
130 INPUT INTERFACE
132 PRIMARY STORAGE DEVICE
140 DATABASES
134 STORAGE INTERFACE
133 BACKUP STORAGE DEVICE

FIG. 1

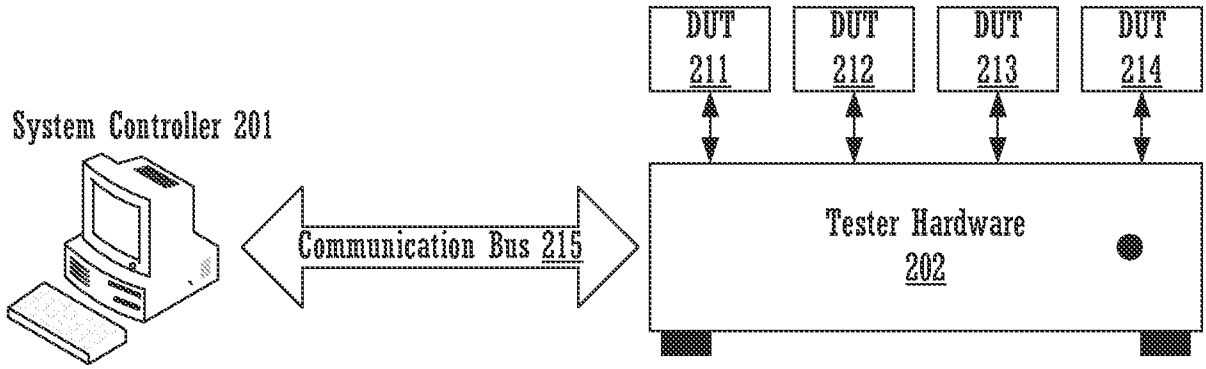
DUT 211
DUT 212
DUT 213
DUT 214
System Controller 201
Communication Bus 215
Tester Hardware 202

FIG. 2A

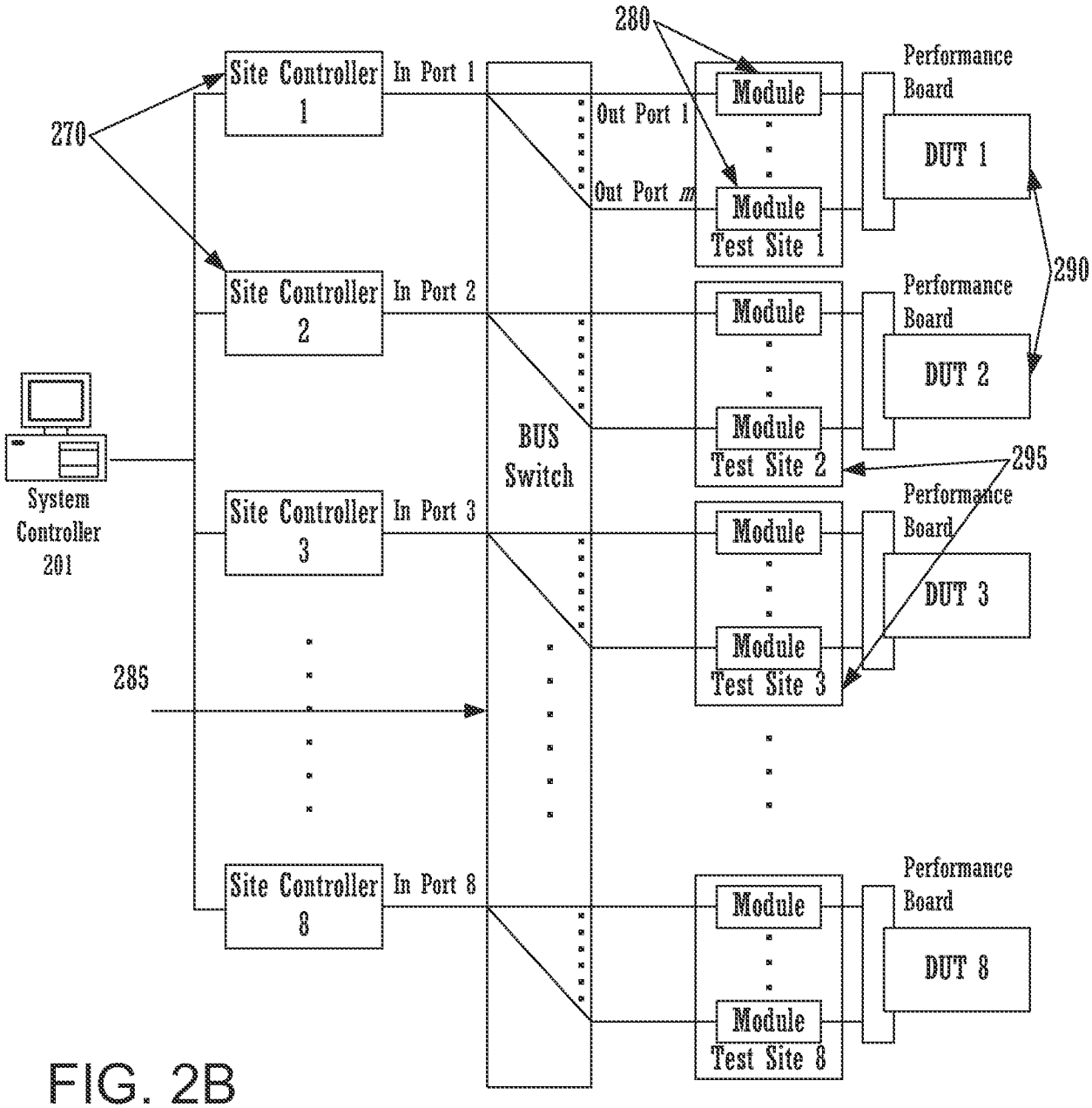
280
270
Site Controller 1
In Port 1
Out Port 1
Out Port m
Module
Test Site 1
Performance Board
DUT 1
290
Site Controller 2
In Port 2
Module
Test Site 2
Performance Board
DUT 2
295
System Controller 201
BUS Switch
Site Controller 3
In Port 3
Module
Test Site 3
Performance Board
DUT 3
285
Site Controller 8
In Port 8
Module
Test Site 8
Performance Board
DUT 8

FIG. 2B

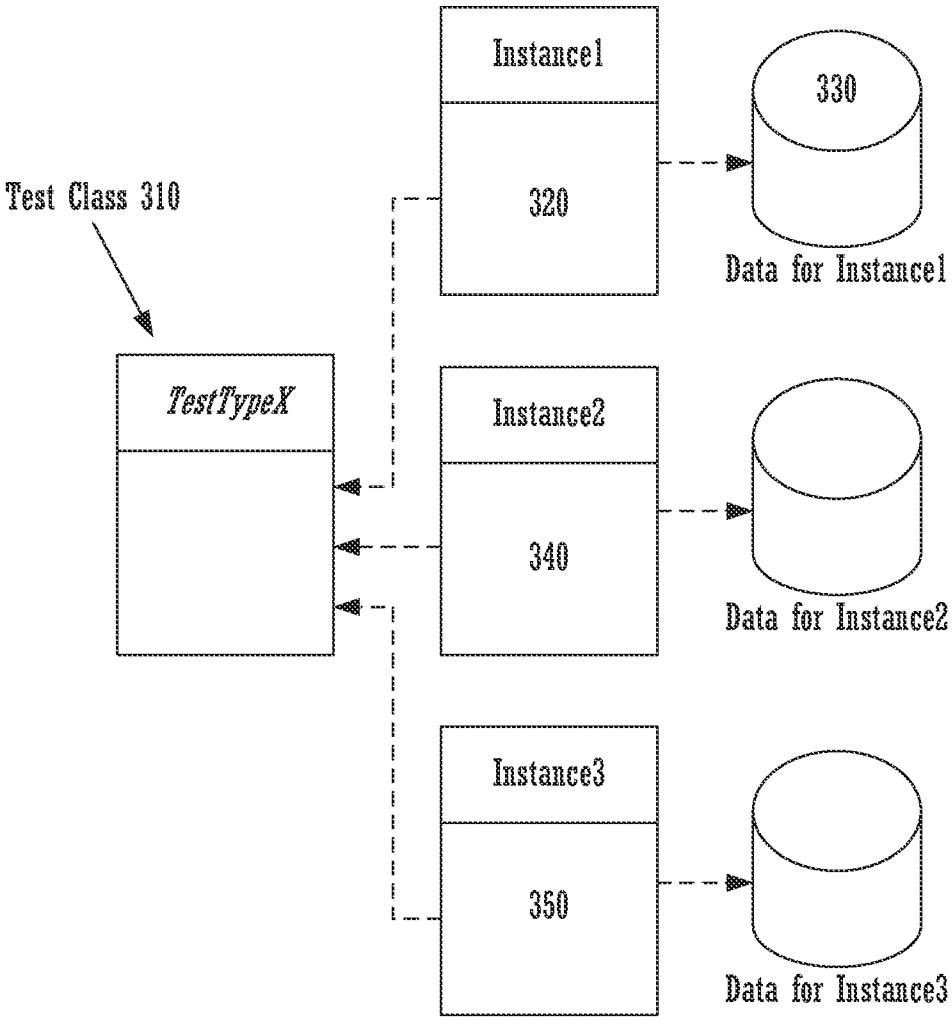
Test Class 310
TestTypeX
Instance1
320
330
Data for Instance1
Instance2
340
Data for Instance2
Instance3
350
Data for Instance3

FIG. 3

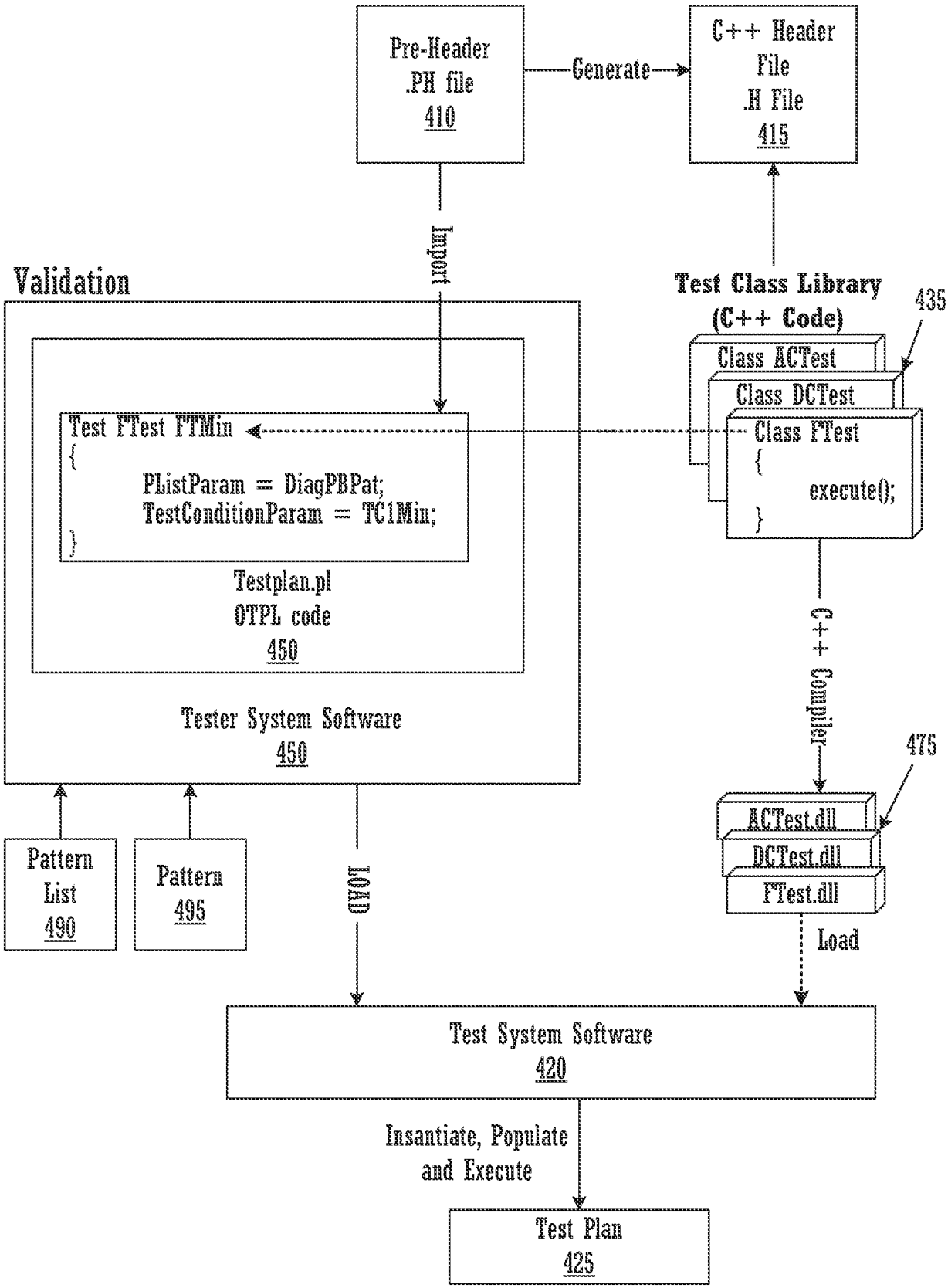
Pre-Header
.PH file
410
Generate
C++ Header
File
.H File
415
Import
Validation
Test Class Library
(C++ Code)
435
Class ACTest
Class DCTest
Class FTest
{
execute();
}
Test FTest FTMin
{
PListParam = DiagPBPat;
TestConditionParam = TC1Min;
}
Testplan.pl
OTPL code
450
Tester System Software
450
C++ Compiler
475
ACTest.dll
DCTest.dll
FTest.dll
Load
Pattern
List
490
Pattern
495
LOAD
Test System Software
420
Insantiate, Populate
and Execute
Test Plan
425

FIG. 4

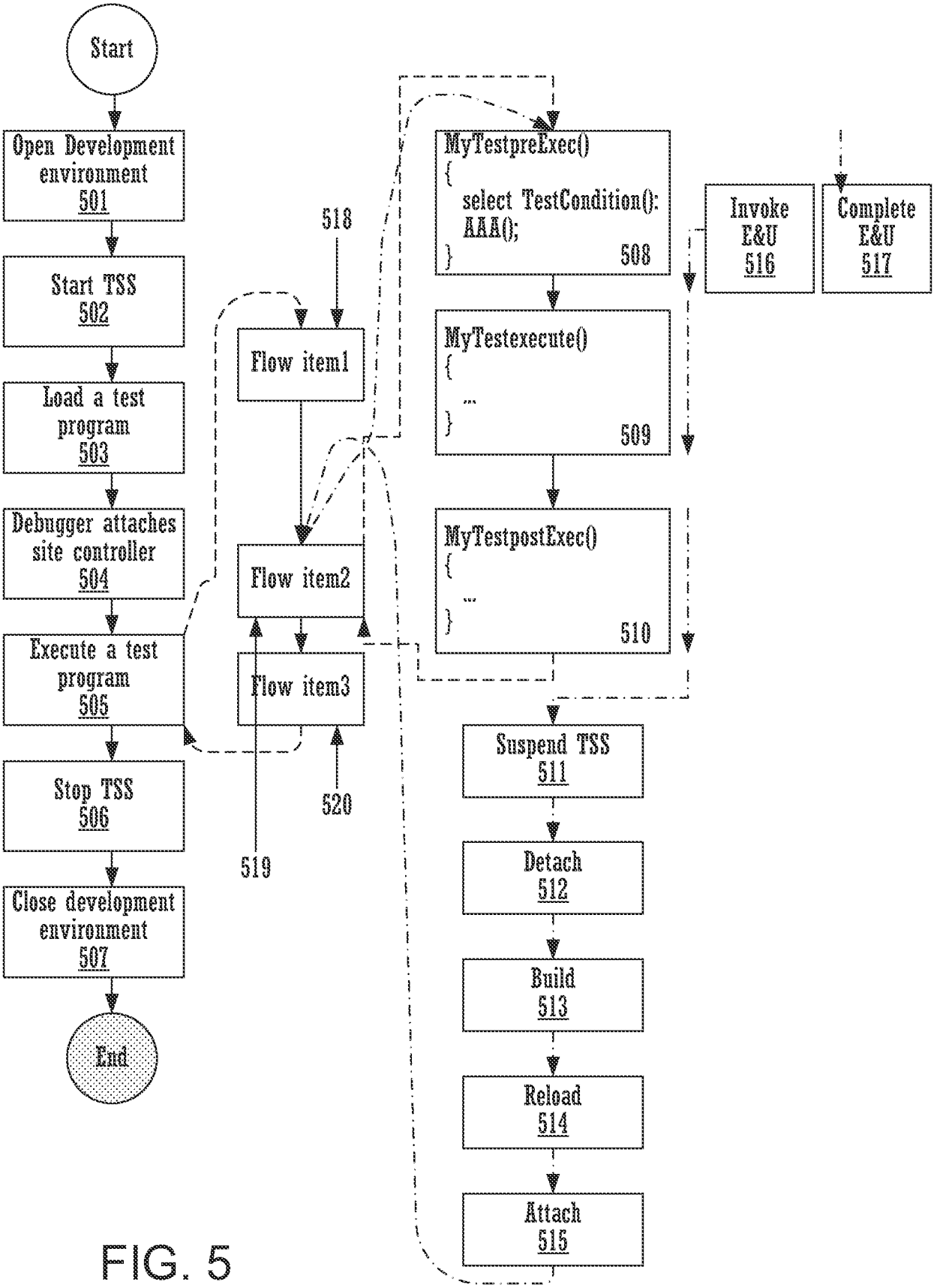
Start
Open Development environment 501
Start TSS 502
Load a test program 503
Debugger attaches site controller 504
Execute a test program 505
Stop TSS 506
Close development environment 507
End
518
Flow item1
Flow item2
Flow item3
519
520
MyTestpreExec()
{
select TestCondition():
AAA();
}
508
MyTestexecute()
{
...
}
509
MyTestpostExec()
{
...
}
510
Invoke E&U 516
Complete E&U 517
Suspend TSS 511
Detach 512
Build 513
Reload 514
Attach 515

FIG. 5

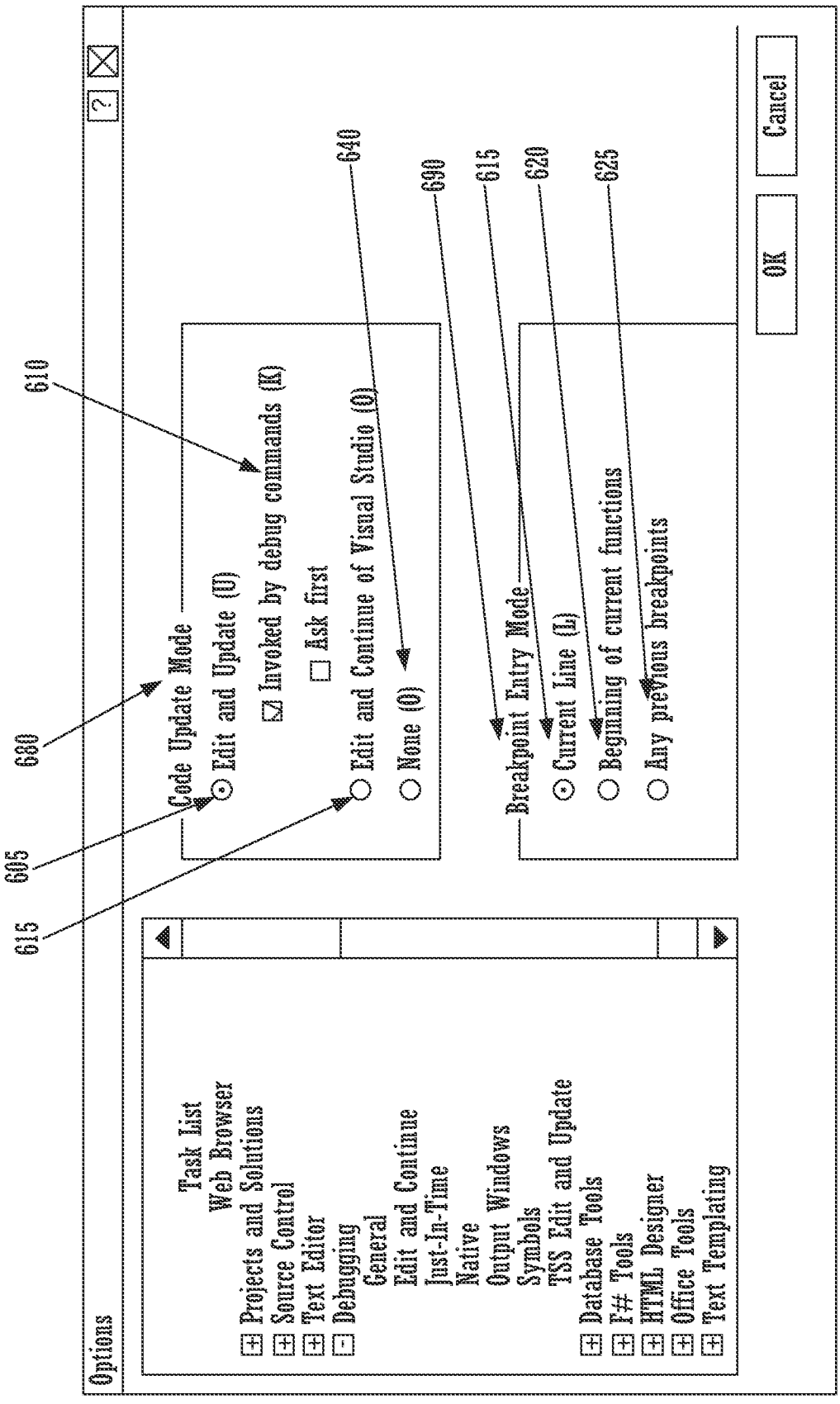
Options
Task List
Web Browser
Projects and Solutions
Source Control
Text Editor
Debugging
General
Edit and Continue
Just-In-Time
Native
Output Windows
Symbols
TSS Edit and Update
Database Tools
F# Tools
HTML Designer
Office Tools
Text Templating
Code Update Mode
Edit and Update (U)
Invoked by debug commands (K)
Ask first
Edit and Continue of Visual Studio (O)
None (O)
Breakpoint Entry Mode
Current Line (L)
Beginning of current functions
Any previous breakpoints
OK
Cancel
610
680
605
615
640
690
615
620
625

FIG. 6

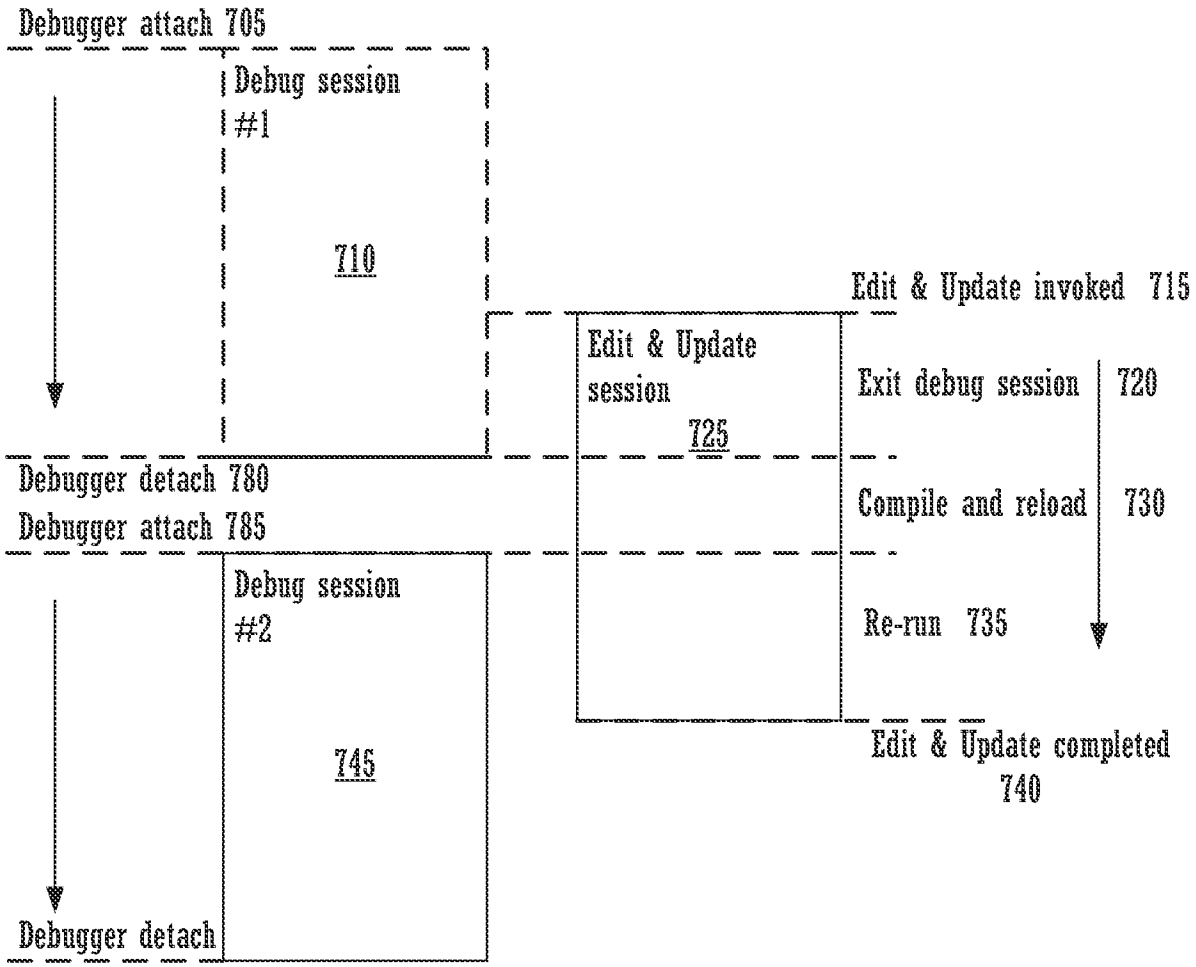
Debugger attach 705
Debug session #1
710
Edit & Update invoked 715
Edit & Update session
725
Exit debug session 720
Debugger detach 780
Compile and reload 730
Debugger attach 785
Debug session #2
Re-run 735
745
Edit & Update completed 740
Debugger detach

FIG. 7

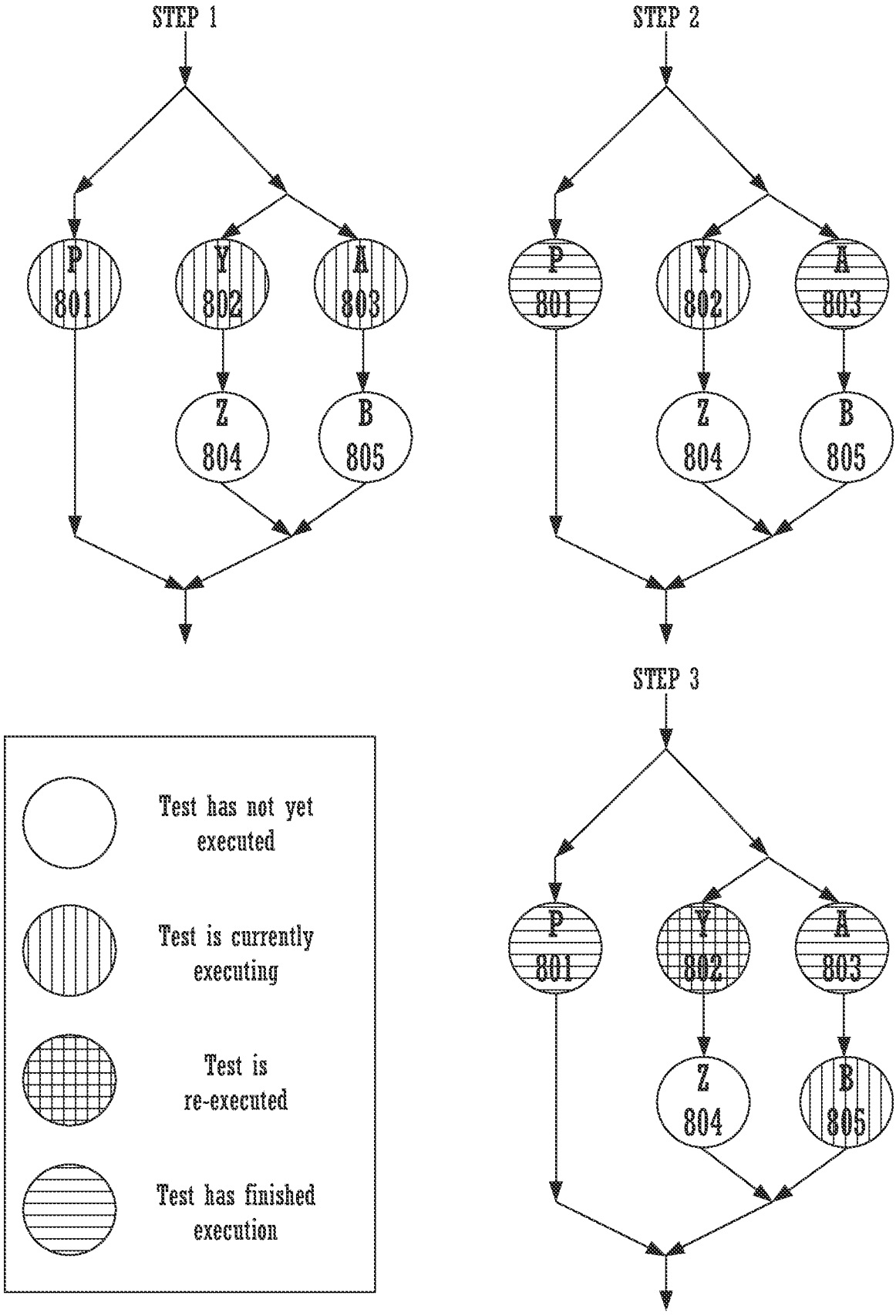
STEP 1
P
801
Y
802
A
803
Z
804
B
805
STEP 2
P
801
Y
802
A
803
Z
804
B
805
STEP 3
P
801
Y
802
A
803
Z
804
B
805
Test has not yet executed
Test is currently executing
Test is re-executed
Test has finished execution

FIG. 8

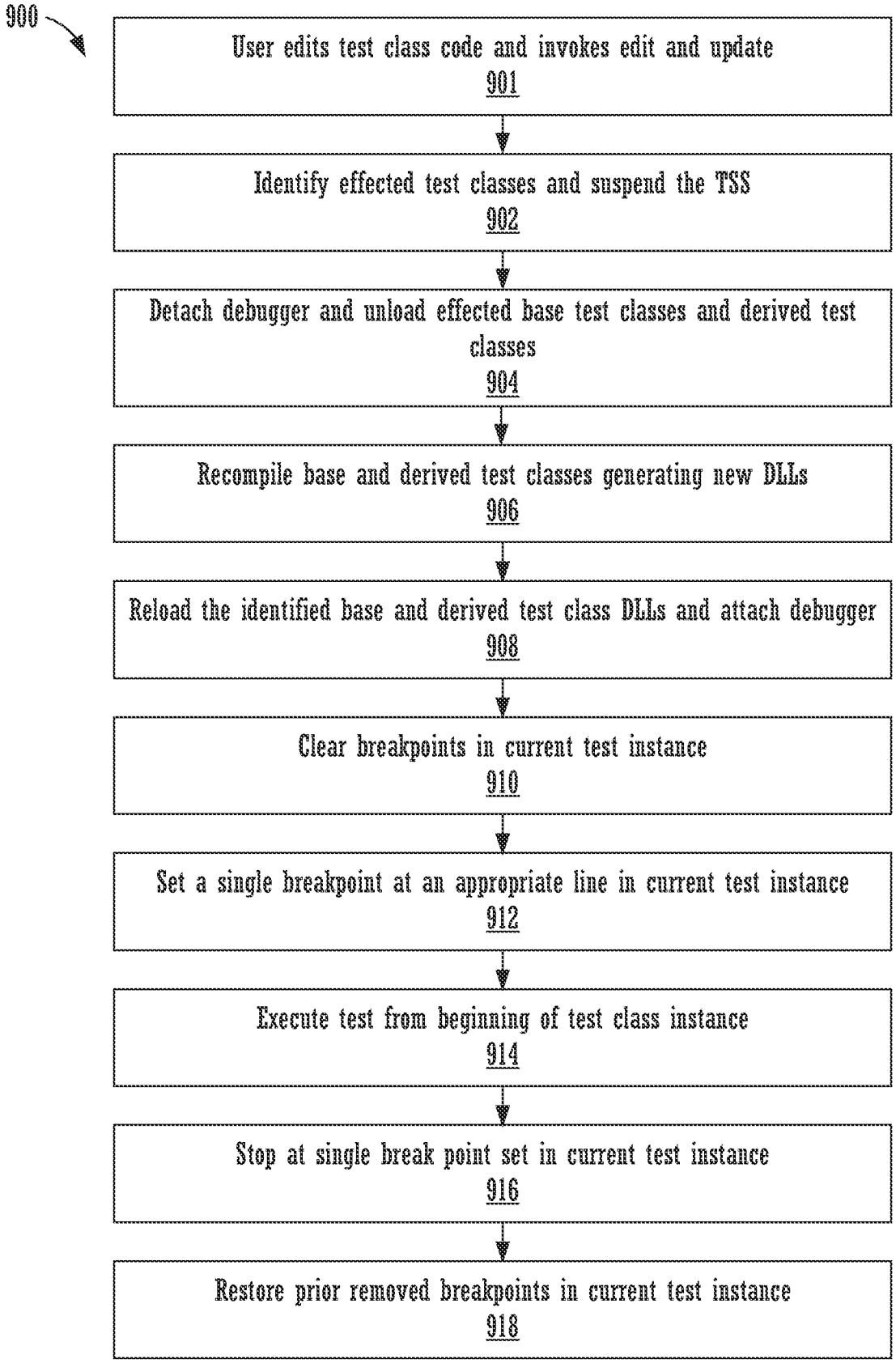
900
User edits test class code and invokes edit and update
901
Identify effected test classes and suspend the TSS
902
Detach debugger and unload effected base test classes and derived test classes
904
Recompile base and derived test classes generating new DLLs
906
Reload the identified base and derived test class DLLs and attach debugger
908
Clear breakpoints in current test instance
910
Set a single breakpoint at an appropriate line in current test instance
912
Execute test from beginning of test class instance
914
Stop at single break point set in current test instance
916
Restore prior removed breakpoints in current test instance
918

FIG. 9

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2014/015963

A. CLASSIFICATION OF SUBJECT MATTER

G01R 31/28(2006.01)i

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)
G01R 31/28; G06F 11/26; G06F 9/44; G06F 11/00; H02H 3/05

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & keywords: break point, debug, DUT, compile

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5675803 A (PREISLER et al.) 07 October 1997 See abstract, column 8, line 18 - column 11, line 11, claims 1-5 and figures 2-7.	1-20,23
A	US 2009-0204849 A1 (SHIMURA) 13 August 2009 See abstract, paragraphs [0008]-[0009], [0053], [0085]-[0087], claim 1 and figures 5-7.	1-20,23
A	US 2008-0126865 A1 (LEE) 29 May 2008 See paragraphs [0040]-[0054], claims 1-6 and figures 3-4.	1-20,23
A	US 2005-0120274 A1 (HAGHIGHAT et al.) 02 June 2005 See abstract, paragraphs [0016]-[0021] and figures 1-4.	1-20,23
A	US 2005-0039079 A1 (HIGASHI et al.) 17 February 2005 See abstract, claims 1-6 and figures 1-6.	1-20,23

☐ 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 application or patent 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
11 June 2014 (11.06.2014)	**11 June 2014 (11.06.2014)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-701,
Republic of Korea
Facsimile No. +82-42-472-7140

Authorized officer
KANG, Sung Chul
Telephone No. +82-42-481-8405

Form PCT/ISA/210 (second sheet) (July 2009)

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2014/015963

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 05675803 A	07/10/1997	EP 0665496 A1	02/08/1995
		EP 0665496 B1	07/07/1999
		EP 0699996 A1	06/03/1996
		JP 08-036488 A	06/02/1996
		JP 08-179940 A	12/07/1996
		US 05581697 A	03/12/1996
US 2009-0204849 A1	13/08/2009	JP 2009-193109 A	27/08/2009
		JP 5022262 B2	12/09/2012
		US 8010839 B2	30/08/2011
US 2008-0126865 A1	29/05/2008	KR 10-1137569 B1	19/04/2012
		KR 10-2008-0000476 A	02/01/2008
US 2005-0120274 A1	02/06/2005	None	
US 2005-0039079 A1	17/02/2005	AT 390637 T	15/04/2008
		CN 100456043 C0	28/01/2009
		CN 100489554 C	20/05/2009
		CN 100541218 C	16/09/2009
		CN 100580473 C	13/01/2010
		CN 100585422 C	27/01/2010
		CN 101231325 A	30/07/2008
		CN 101231325 B	31/08/2011
		CN 101231325 C0	30/07/2008
		CN 1756960 A	05/04/2006
		CN 1756960 C0	12/11/2008
		CN 1768275 A	03/05/2006
		CN 1768275 C0	03/05/2006
		CN 1774642 A	17/05/2006
		CN 1784609 A	07/06/2006
		CN 1784609 B	23/02/2011
		CN 1784609 C0	07/06/2006
		CN 1981200 A	13/06/2007
		CN 1981200 C0	13/06/2007
		CN 1981202 A	13/06/2007
		CN 1981202 C0	13/06/2007
		CN 1981203 A	13/06/2007
		CN 1981203 C0	13/06/2007
		CN 1989417 A	27/06/2007
		CN 1989417 B	16/03/2011
		CN 1989417 C0	27/06/2007
		CN 1997908 A	11/07/2007
		CN 1997908 C0	11/07/2007
		CN 1997909 A	11/07/2007
		CN 1997909 B	10/11/2010
		CN 1997909 C0	11/07/2007
		DE 602004012714 D1	08/05/2008
		DE 602004012714 T2	16/04/2009

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2014/015963

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
		EP 1592975 A1	09/11/2005
		EP 1592975 B1	26/03/2008
		EP 1592976 A1	09/11/2005
		EP 1592976 B1	16/01/2008
		EP 1610135 A1	28/12/2005
		EP 1610135 B1	08/10/2008
		EP 1610135 B8	04/03/2009
		EP 1610136 A1	28/12/2005
		EP 1610136 A4	17/05/2006
		EP 1610136 B1	11/07/2007
		EP 1756601 A2	28/02/2007
		EP 1756601 B1	09/12/2009
		EP 1756602 A2	28/02/2007
		EP 1756602 B1	09/12/2009
		EP 1756603 A1	28/02/2007
		EP 1756603 B1	05/08/2009
		EP 1756604 A1	28/02/2007
		EP 1756604 B1	19/08/2009
		EP 1756605 A1	28/02/2007
		EP 1756605 B1	07/11/2007
		EP 1756606 A1	28/02/2007
		EP 1756606 B1	12/08/2009
		EP 1767955 A1	28/03/2007
		EP 1767955 B1	01/10/2008
		EP 1870724 A1	26/12/2007
		EP 1884789 A2	06/02/2008
		EP 1884789 A3	13/02/2008
		EP 1884789 B1	29/07/2009
		JP 2006-053160 A	23/02/2006
		JP 2006-518460 A	10/08/2006
		JP 2006-520947 A	14/09/2006
		JP 2007-052028 A	01/03/2007
		JP 2007-057541 A	08/03/2007
		JP 2007-256295 A	04/10/2007
		JP 2007-279050 A	25/10/2007
		JP 2007-518967 A	12/07/2007
		JP 2007-528992 A	18/10/2007
		JP 2007-528993 A	18/10/2007
		JP 2007-528994 A	18/10/2007
		JP 2008-519247 A	05/06/2008
		JP 2008-519248 A	05/06/2008
		JP 2009-008683 A	15/01/2009
		JP 3735636 B2	18/01/2006
		JP 3748269 B2	22/02/2006
		JP 3890079 B1	07/03/2007
		JP 3911007 B1	09/05/2007
		JP 3939336 B2	04/07/2007
		JP 3954639 B2	08/08/2007
		JP 4332179 B2	16/09/2009
		JP 4332200 B2	16/09/2009

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2014/015963

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
		JP 4516961 B2	04/08/2010
		JP 4608516 B2	12/01/2011
		JP 4608517 B2	12/01/2011
		JP 4704178 B2	15/06/2011
		KR 10-2005-0099626 A	14/10/2005
		KR 10-2005-0101216 A	20/10/2005
		KR 10-2005-0113271 A	01/12/2005
		KR 10-2005-0113273 A	01/12/2005
		KR 10-2007-0014206 A	31/01/2007
		KR 10-2007-0017535 A	12/02/2007
		KR 10-2007-0020247 A	20/02/2007
		KR 10-2007-0020300 A	20/02/2007
		KR 10-2007-0023762 A	28/02/2007
		KR 10-2007-0035507 A	30/03/2007
		TW I315406 A	01/10/2009
		TW I315406 B	01/10/2009
		TW I350463 I	11/10/2011
		TW I350965 I	21/10/2011
		TW I389033 I	11/03/2013
		US 2004-0193990 A1	30/09/2004
		US 2004-0210798 A1	21/10/2004
		US 2004-0225459 A1	11/11/2004
		US 2004-0255216 A1	16/12/2004
		US 2004-225465 A1	11/11/2004
		US 2005-0022087 A1	27/01/2005
		US 2005-0154550 A1	14/07/2005
		US 2005-0154551 A1	14/07/2005
		US 2005-0261855 A1	24/11/2005
		US 2005-0262412 A1	24/11/2005
		US 2005-0262414 A1	24/11/2005
		US 2008-0010524 A1	10/01/2008
		US 2008-0016396 A1	17/01/2008
		US 2010-0192135 A1	29/07/2010
		US 7184917 B2	27/02/2007
		US 7197416 B2	27/03/2007
		US 7197417 B2	27/03/2007
		US 7209851 B2	24/04/2007
		US 7210087 B2	24/04/2007
		US 7272765 B2	18/09/2007
		US 7290192 B2	30/10/2007
		US 7430486 B2	30/09/2008
		US 7437261 B2	14/10/2008
		US 7460988 B2	02/12/2008
		US 8255198 B2	28/08/2012
		WO 2004-072669 A1	26/08/2004
		WO 2004-072670 A1	26/08/2004
		WO 2004-088339 A1	14/10/2004
		WO 2004-090562 A1	21/10/2004
		WO 2005-114235 A2	01/12/2005
		WO 2005-114235 A3	20/04/2006

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2014/015963

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
		WO 2005-114237 A1	01/12/2005
		WO 2005-114238 A1	01/12/2005
		WO 2005-114239 A1	01/12/2005
		WO 2005-114240 A1	01/12/2005
		WO 2005-114241 A2	01/12/2005
		WO 2005-114241 A3	20/04/2006