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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
1 July 2010 (01.07.2010)

PCT

(10) International Publication Number
WO 2010/074860 A2

(51) International Patent Classification:
*G06F 11/36* (2006.01) *G06F 11/28* (2006.01)

(21) International Application Number:
PCT/US2009/065324

(22) International Filing Date:
20 November 2009 (20.11.2009)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
12/335,739 16 December 2008 (16.12.2008) US

(71) Applicant *(for all designated States except US)*: **MICROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(72) Inventors: **STALL, Jonathon, Michael**; c/o Microsoft Cororation, One Microsoft Way, Redmond, Washington 98052-6399 (US). **CRIDER, Anthony, L.**; c/o Microsoft Cororation, One Microsoft Way, Redmond, Washington 98052-6399 (US). **ZINKOVSKY, Igor, A.**; c/o Microsoft Cororation, One Microsoft Way, Redmond, Washington 98052-6399 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, 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, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

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

**Declarations under Rule 4.17**:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

**Published**:

— *without international search report and to be republished upon receipt of that report (Rule 48.2(g))*

(54) Title: TRANSFORMING USER SCRIPT CODE FOR DEBUGGING

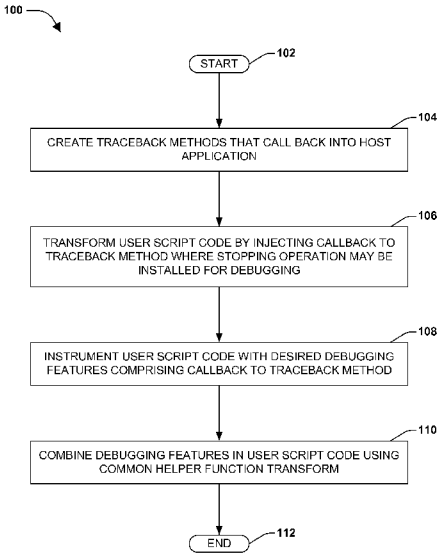


FIG. 1

(57) Abstract: User script code that is developed to be run in a host application, for example, as a macro can be transformed into debuggable code so that the host application may continue to operate during a debugging stop operation. Traceback methods can be created that call back into the host application to allow the host application to cooperatively operate and update its user-interface. The user script code can be transformed by injecting callbacks to the traceback methods at respective locations in the code where a stopping operation may be installed during debugging. Further, two or more debugging features can be combined into a single user script code transform using an iterator pattern function.

WO 2010/074860 A2

# TRANSFORMING USER SCRIPT CODE FOR DEBUGGING

## BACKGROUND

**[0001]** Users of computer program applications often find a need to perform specific tasks in the application, sometimes over and over again. For example, a company managing their financial issues may wish perform certain calculations on data populating a spreadsheet in an application. Performing these tasks may be supported by a host application, but applications may also allow a developer/user to create custom solutions to automate desired tasks.

**[0002]** A user of an application or a developer can create macros that perform supported functions in the application, thereby automating a desired task. Macros can be created by recording a series of actions the user performs in the application, or can be developed in source code that is supported by the host application. Developers/users may wish to debug a macro, either before use or after detecting undesirable operations of the macro in the host application. Host applications may run on a platform that comprises an implementation of a programming language framework in an integrated development environment (IDE), which enables a user/developer to write source code, compile the code, and debug the code to be run in the host application.

## SUMMARY

**[0003]** This Summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key factors or essential features of the claimed subject matter, nor is it intended to be used to limit the scope of the claimed subject matter.

**[0004]** During debugging of their user script code a developer often inserts breakpoints where they can inspect the test environment to determine whether it is functioning as expected. Further, at breakpoints, a developer may wish to inspect variables, step to different functions without executing the code in between, edit code and continue execution, review exception handling, amongst other things. Currently, during most new debugging operations (e.g., using C# and .NET scripting), when the debugger hits a breakpoint in the application (debuggee) it causes the debuggee to freeze until the debugger is restarted (e.g., when restart is selected by the user). Some older IDE platforms for host applications allow the host application to continue to respond when a breakpoint is hit, which may be desirable. However, if a host application was move from an older scripting platform to a

new scripting platform, a user may not be able to have that host application continue to respond at a debugging breakpoint.

**[0005]** As set forth herein, techniques and systems are provided for enabling debuggability of user script code developed for running in a host application by transforming the user script code into debuggable code. For example, the user script code can be instrumented with script code that can make the user script code debuggable, allowing the debugger to get debugging information from the user code instead of from a debugging API that may impact the host application.

**[0006]** In one embodiment, traceback methods can be created that call back into the host application, which may allow the host application to cooperatively operate and update its user-interface and to appear to continue functioning, even at a debugging stop operation (e.g., a breakpoint). Further, the user script code can be transformed by injecting a callback to a traceback method at one or more location in the user script code where a stopping operation may be inserted for debugging. Additionally, debuggability features may be combined in the user script code by transforming the user script code with a common helper function (e.g., an iterator-type function), for example, instead of transforming the code for each debuggability feature desired by the user (e.g., stack frame management, variable inspection, set-next statements, etc.).

**[0007]** To the accomplishment of the foregoing and related ends, the following description and annexed drawings set forth certain illustrative aspects and implementations. These are indicative of but a few of the various ways in which one or more aspects may be employed. Other aspects, advantages, and novel features of the disclosure will become apparent from the following detailed description when considered in conjunction with the annexed drawings.

## DESCRIPTION OF THE DRAWINGS

**[0008]** Fig. 1 is a flow diagram of an exemplary method whereby user script code that has been developed for running in a host application can be enabled for debuggability

**[0009]** Fig. 2 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0010]** Fig. 3 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0011]** Fig. 4 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0012]** Fig. 5 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0013]** Fig. 6 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0014]** Fig. 7 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0015]** Fig. 8 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0016]** Fig. 9 is an illustration of a transformation of exemplary script code utilizing techniques described herein.

**[0017]** Fig. 10 is a block diagram of an exemplary system for transforming user script code, developed for running in a host application, into debuggable code.

**[0018]** Fig. 11 is an illustration of one embodiment of an example implementation of the exemplary systems described herein.

**[0019]** Fig. 12 is an illustration of an exemplary computer-readable medium comprising processor-executable instructions configured to embody one or more of the provisions set forth herein.

**[0020]** Fig. 13 illustrates an exemplary computing environment wherein one or more of the provisions set forth herein may be implemented.

## DETAILED DESCRIPTION

**[0021]** The claimed subject matter is now described with reference to the drawings, wherein like reference numerals are used to refer to like elements throughout. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the claimed subject matter. It may be evident, however, that the claimed subject matter may be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to facilitate describing the claimed subject matter.

**[0022]** Fig. 1 is a flow diagram of an exemplary method 100 whereby user script code that has been developed for running in a host application can be enabled for debuggability. For example, a user of an application that supports development of custom add-ons may write user script code for a custom macro for the application, and may wish to debug the custom macro. In this example, the exemplary method 100 can be used to transform the user script code into debuggable script code.

**[0023]** The exemplary method 100 begins at 102 and involves creating traceback methods that call back into the host application to allow the host application to cooperatively operate and update its user-interface (UI), at 104. In one embodiment, traceback methods can be created in a library of an application execution and development framework (e.g., ".NET," Java), which can then be called by script code executed in the framework. In this embodiment, when called, the traceback methods can allow a host application to appear to continue functioning at a debugging stopping operation by allowing the host application to operate its UI.

**[0024]** Currently, as an example, when a debugger hits a breakpoint in the debuggee (e.g., the host application running the user script code), the debuggee may freeze until it is inspected by the user and the debugger is restarted. In this example, the debuggee typically freezes because it is being controlled by the debugger, which stops the host application at a breakpoint, inspects the data and merely restarts the debuggee when a user commands it to continue. Traditionally, debugging breakpoints stop a thread in the application, for example, which blocks a message pump (e.g., message pump picks up control commands and executes event associated with command) from responding to commands in the UI.

**[0025]** However, a traceback method can be used to transfer control back to the host application, for example, at a breakpoint. The host application may continue to pump messages from the UI (e.g., or console inputs for a console hosts) instead of having the debugger call the operating system to park the thread at an opcode for the breakpoint, as in traditional debugging breakpoints. In this example, the thread can be parked in the traceback method, which transfers control back to the host application.

**[0026]** At 106, in the exemplary method 100, the script code created by a user for use in the host application can be transformed by injecting a callback, which calls to a traceback method, at one or more locations in the user script code where a stopping operation may be installed for debugging. Callbacks to a traceback method can be injected at respective locations where a potential breakpoint may be used in debugging. In this way, for example, whenever a breakpoint is encountered during debugging of the user script code, the traceback method can be called, which allows for the host application to continue functioning during the breakpoint.

**[0027]** At 108, the user script code is instrumented with desired script code debugging features that comprise a callback to a traceback method. For example, a user that develops script code to run as a macro in a host application may wish to enable desired debugging features for the user script code. In this example, the user's desired debugging features can

be instrumented into the user script code during compilation of the code (e.g., by a framework's high-level language compiler or by a dynamic runtime environment of the framework), which can transform the user script code into debuggable code for the desired features.

**[0028]** In one embodiment, a host application's macro development platform may present a user with a menu of features that can be instrumented into the user's code. In this way, for example, the user can select merely those debugging features desired, and the user script code can be transformed by instrumenting the user script code with those selected debugging features.

**[0029]** At 110, in the exemplary method 100, two or more separate script code debugging features can be combined in the user script code by using a common helper function transform. For example, debugging features that a user wishes to include in their user script code may be unified together in a common helper transform of the code, which can support respective debugging features. In one embodiment, for example, where more than one debugging feature may be instrumented into the user script code at a same location, they can be combined in a common helper function transformation of the code that provides support for the respective features.

**[0030]** Having combined debugging features, the exemplary method 100 ends at 112.

**[0031]** In one aspect, transforming the user script code can be high-level language agnostic. In one embodiment, the user script code can be transformed by a language compiler that is part of an application execution and development framework, which compiles a high-level language into an intermediate language (IL) utilized by the framework. In another embodiment, the script code transforms may be done in the framework's dynamic runtime environment (e.g., .NET just-in-time compilation service in the common language runtime), which compiles the framework's IL into machine language to be executed. In these embodiment, the transforms may be language agnostic, as transformations are done during compiling using the framework's compilation services.

**[0032]** In another aspect, calls to traceback methods are injected into the user script code at those locations where a stopping operation (e.g., a breakpoint, stepping operation, asynchronous break, etc.) may be used by a debugger. In one embodiment, in this aspect, a user script code transformation can inject calls to a traceback method at respective potential stopping points where arguments to the traceback method can include a context of the stopping operation.

**[0033]** Fig. 2 is an illustration of exemplary script code 200 where calls to traceback methods are injected at potential stopping points in the code. Script code 202 written by a user (e.g., for use as a macro in a host application) is comprised of three lines of code 206, lines A, B, and C. During debugging, a debugger may utilize breakpoints before each of the lines of code 206. In this example, the user script code 202 can be transformed 204 (e.g., by a compiler) by injecting a callback to a traceback method 208 before the respective lines of code 206. Here, calls to tracebacks one, two, and three 208, have been injected that can identify a location of potential breakpoints.

**[0034]** In one embodiment, for example, when the debugger appears to have stopped the debuggee at a breakpoint, a thread may not actually be stopped at a hard-mode breakpoint. Instead, in this example, the thread can be parked inside the called traceback method implemented by a host application, at a cooperative control delegator (e.g., a message pump) that allows the host to continue to function. Further, in this example, the thread may not resume from the traceback method until after the debugger continues from the breakpoint.

**[0035]** Additionally, in this embodiment, tracebacks can be a no-operation (nop), so that they have little impact on the executing code. For example, if a debugger requests a stopping operation (e.g., a breakpoint) the traceback method can detect the requested stop appropriately. However, if there is no requested stopping operation from the debugger, no operation is performed by the traceback method.

**[0036]** In another aspect, various user script code transformations may be performed to enable debugging features. It will be appreciated that, while the following embodiments and examples identify several common debugging features for script code, the techniques and systems, described herein, are not limited to any particular debugging features. Those skilled in the art may devise alternate debugging features that can be instrumented into a user script code, developed for running in a host application, by transforming the user script code into debuggable code.

**[0037]** Fig. 3 is an illustration of exemplary script code 300 where user script code can be transformed to perform stepping during debugging of the script code. An example line of user script code 302 can comprise locations that can be stepped to or a breakpoint may be set during debugging: one may stop at a beginning 304 of the user script code 302 for a source-level breakpoint; after stepping out of Alpha, before stepping into Beta 306; after stepping out of Beta; and after stepping out of Foo().

**[0038]** The example line of user script code 302 can be transformed into debuggable code 312 that comprises calls to traceback methods at the respective stepping or stopping points in the user script code. Therefore, a call to traceback one 314 can be injected at the beginning of the line 304; a call to traceback two 316 can be injected after Alpha and before Beta 306; a call to traceback three can be injected after Beta 308; and a call to traceback four can be injected after Foo() 310.

**[0039]** Traditionally, stepping through script code during debugging is supported by a CPU using flags and executing single instructions. In this embodiment, stack steps of the respective tracebacks are known, depending on a type of stepping performed (e.g., step-in, step-out, step-over), a next traceback is known based on its designation. For example, a step-in can merely be stopping at a next traceback (e.g., from traceback two to three), regardless of the function that comprises the traceback. Further, in this example, a step-over may be stopping a next traceback and a current (or less) stack level. Additionally, in this example, a step-out can be stopping at a next traceback at a shallower stack level.

**[0040]** Fig. 4 is an illustration of transformed exemplary script code 400 that may allow a function to maintain its own callstack, comprising script frames from the user code, for a debugger. In this example, script code for a function Foo() 402 has been transformed, whereby a thread's debugger stack, $thread 406, can be maintained by the function Foo(), allowing it to push or pop a debugger frame 404. The transformation utilizes a call to a traceback method 408 to allow continued functionality of the host application while the debugger pushes or pops frames.

**[0041]** In one embodiment, the thread's debugger stack, $thread 406, can be stored in a thread-local storage. In another embodiment, the thread's debugger stack, $thread 406, can be passed as a hidden parameter to respective user-code functions. In this way, for example, a debugger's callstack window may be an inspection operation on user data in the thread's debugger stack, $thread 406.

**[0042]** Fig. 5 is an illustration of exemplary script code 500 where user script code can be transformed to allow a single function to be split into a separate header and body. In this example, user script code 502 for a function foo() comprises the function's header 506 and its body 508. The user script code 502 can be transformed into debuggable script code 504 that splits the function into its header 506 and body 508, for example, during compilation.

**[0043]** In the debuggable script code 504, the function header 506 has been transformed into a function that identifies the header by a reference "foo" 512; and the function body 508 has been transformed into a function foo_body 514. A reference has been created for a

global table 510 that links the header function 512 with the body function 514. This may allow the function body 514 to be stored in the global table and be updated, for example. Further, in this example, the header function 512 can be used to fetch a latest version of the corresponding body from the global table.

**[0044]** In one embodiment, various transforms may be applied to a function body. A function body may be swapped out for editing purposes in an "edit-and-continue" debugging feature, for example. In this example, a first function body can be swapped out and a set-next statement may be used to map from the first function body to a second function body. In another example, the function body may be swapped out for deoptimization purpose, while other functions can continue to have a stable reference to the function header.

**[0045]** In another embodiment, common debugging infrastructure may be stored and shared in a function header, split from its body. For example, infrastructure code stored in the function header could comprise notification hooks used by a debugger. In another example, infrastructure code stored in the function header could comprise drivers for a switch table that may be used for set-next statements in debugging.

**[0046]** Fig. 6 is an illustration of exemplary script code 600 where user script code can be transformed to allow for value inspection during debugging. For example, a function may publish it local variable to a frame object, which can be done in different ways. In this example, a function foo() 602 comprises a string variable 'x', and statements for writing and reading the local variable.

**[0047]** In one embodiment 604, the locals can be hoisted into a closure or dictionary object 608. In one example, in this embodiment, hoisting the locals into a dictionary object may adjust the locals storage and may force them to be on a heap. In 604, statements comprise calls to traceback methods for writing 610 and reading 612 the local variables.

**[0048]** In another embodiment 606, addresses of locals may be described via a customized display frame 614. In one example, in this embodiment, the locals could be allowed to live on a stack comprising the customized frame 614, which may have less impact on the locals storage. For the function foo() a new frame can be created 616 that comprises unverifiable pointer code. Statements for writing 618 and reading 620 the local respectively comprise calls traceback methods, allowing for a host application to continue to operate its UI during debugging, for example.

**[0049]** In both embodiments, variable homes are described in a display object 610, 612, 618, 620, which can be available in a traceback for the debugger. As an example, the locals

may not need to be "spilled" to a display object until the function reaches a traceback, allowing the host application to continue to operate its UI during debugging. In 604, the locals can be "spilled" to the stack to be preserved across traceback function calls, for example.

**[0050]** Fig. 7 is an illustration of exemplary script code 700 where user script code can be transformed to allow for use of "set-next" statements during debugging. In one embodiment, using a set-next statement a debugger can set a next active statement in a current frame without executing user code in between. In this embodiment, a source and target statement are in a same function, and either can be before or after the other. For example, if a thread is parked at a traceback for line B 710, a debugger user may be able to move the thread to a traceback for line A 708, without executing user code in between.

**[0051]** This type of set-next statement can be implemented by injecting a switch table 714 into the function, as illustrated in the exemplary script code 700. In this example, if a debugger user does a set-next statement while in a traceback 708, 710, 712, the debugger can set the pseudo local "$nextLine" 714 to a target line and the appropriate traceback can return "true." In this example, this can cause a control for a pointer to jump to the next switch table (at a switch label) 714 and then to an appropriate line 702, 704, 706.

**[0052]** It will be appreciated the above embodiment is merely one example of implementing switch tables to utilize set-next statements for debugging. The techniques and systems, described herein, are not limited to the above embodiment. For example, C# "iterators" or "generators" in Python may also be implemented with a switch table. In this example, a user code transform may be able to leverage converting the user code function to an iterator as a means of getting a switch table. Those skilled in the art may devise alternate means for utilizing switch tables for set-next functions in debugging.

**[0053]** Fig. 8 is an illustration of exemplary script code 800 where user script code can be transformed to allow for exception handling during debugging. In one embodiment, an explicit "throw" keyword can be used for a synchronous exception, as in the exemplary code at 802. In this embodiment, calls to traceback methods can be inserted for the synchronous exceptions, so that the "throw" exception example is transformed into 808, where a traceback is called for the explicit exception $t.

**[0054]** In another embodiment, asynchronous exceptions can be made synchronous using explicit checks, such as a divide-by-zero or null reference may cause an asynchronous exception. In this example, the exemplary user code 804 calls for a divide-by function. This code can be transformed 810 so that, if the divide-by value is zero a traceback function

for this exception is called. Otherwise the value can be divided into the original function value.

**[0055]** In another embodiment, script code may call out to non-script code 806. In this embodiment, for example, exceptions in non-script code may not be visible to a script debugger until they enter the script code. In this embodiment, the script code can be transformed 812 so that outgoing calls are wrapped in a try-catch block 812. Further, the transformed code comprises a call to a traceback method for exception handling, as described above.

**[0056]** It will be appreciated that, while several embodiments of transforms of user script to script code comprising common debugging features have been described above, the techniques and systems described herein are not limited to these debugging features. Debugging features are numerous and varied and those skilled in the art may devise alternate debugging features that can be brought into script code by performing user script transforms as described above.

**[0057]** In another aspect, a transform may combine one or more debugging features, such as function header/body splitting, exception handling guards, switch tables, explicit stack frames creation, local variable hoisting, tracebacks, and others into a single transform of the user script code, based on an iterator pattern, such as a common helper function. Fig. 9 is an illustration of exemplary script code 900 where user script code can be transformed to allow for combination of two or more debugging features using an iterator pattern.

**[0058]** The example feature combination transform comprises inserting script code that maps function bodies to a global table of function bodies 902. In this example, the global table "g_functs" maps the function header "foo" with its corresponding body "foo_body." A common helper function 904 can be inserted in the transform that can be shared across different implementations. The helper function 904 can call to the global table 906 for a latest version of a function body for a function header, where a function body may have been edited, for example. Further, the helper function 904 can comprise two or more debugging feature transforms 908 for the user script code, such as tracebacks, set-next statements, and exception handling.

**[0059]** In this example, the function "foo" calls to the common helper function 904, with a string "foo" 910. The "foo_body" 912 is like an initial body but comprises yields where breakpoints may be initiated for debugging. The body 912 is converted into an iterator, where the header 910 drives the iterator and respective yields can become calls to traceback methods before the statement lines of the function. In this way, in this example, the iterator

pattern can be used to combine more than one debugging feature into a user script code transform.

**[0060]** A system may be devised that can enable a user script code, developed for use in a host application, to be debugged, for example, while allowing the host application to operate its user interface (UI). Fig. 10 is a block diagram of an exemplary system 1000 for transforming user script code, developed for running in a host application, into debuggable code. For example, where a user/developer may wish to debug user script code they developed to run as a macro in a host application, the exemplary system 1000 may transform the user script code into debuggable code so that the host application can continue responding to UI events during a stopping operation in debugging.

**[0061]** The exemplary system 1000 comprises traceback methods 1008 that can be configured to call back into the host application to allow the host application to cooperatively operate and update its UI, for example, to appear to continue functioning at a debugging stopping operation. In one embodiment, a traceback method can transfer control back to the host application, which can continue to operate a message pump for performing UI events.

**[0062]** The exemplary system 1000 further comprises a user script code transformation component 1002, which can be configured to transform user script code 1050 developed for running in a host application. The user script code transformation component 1002 comprises a traceback method callback instrumentation component 1004, which can be configured to inject a callback, which calls to a traceback method 1008, at one or more locations in the user script code 1050 where a stopping operation may be installed for debugging. For example, the traceback method callback instrumentation component 1004 may transform user script code 1050 by injecting a call to a traceback prior to respective functions, and/or statements in functions of the user script code 1050.

**[0063]** The user script code transformation component 1002 further comprises a user script code debugging feature combiner 1006 that can be configured to combine two or more script code debugging features in the user script code 1050 by using an iterator pattern transform. For example, if a user/developer of the script code 1050 wished to utilize several debugging features during debugging of the code, such as tracebacks, set-next statements, variable inspection, edit and continue, at others, an iterator pattern may be used to combine these transforms into one transform. In this example, a common helper function can be injected into the code that allows a combination of more than one debugging feature using an iterator pattern. In this way, the user script code transformation component 1002 can

transform user script code 1050 into transformed user script code 1052 that is debuggable, for example, while the host application can continue to function without utilizing cooperation from an operating system's virtual machine.

**[0064]** In one embodiment, the user script code debugging feature combiner can be configured to insert script code that maps function headers to a global table of function bodies. Further, the user script code debugging feature combiner can be configured to insert script code that has respective function headers, having debugging features to be combined, call an iterator pattern function. Additionally, the user script code debugging feature combiner can be configured to retrieve corresponding updated function bodies that have been transformed to include code enabling a debugging feature using the iterator pattern function. In this way, for example, a function that may comprise more than one debugging feature can have a single transform that combines respective debugging features.

**[0065]** In another embodiment, the user script code transformation component 1002 can further comprise a function separation component, which can be configured to compile a function into a separate header and body, and store the function body in a global table comprising function bodies mapped to corresponding function headers. In this way, for example, a function body may be edited during debugging and the body can still be mapped to its corresponding header using the global table.

**[0066]** Fig. 11 is an illustration of one embodiment of an example implementation 1100 of the exemplary system described above. In the exemplary implementation 1100 a host application 1102 (e.g., a spreadsheet application) is running in on an application execution and development framework 1114. Execution and development frameworks 1114 typically have a development library 1112 that may comprise functions that can be called by code written and executed within the framework 1114.

**[0067]** In this example, a user can write script code 1106, such as a macro, that can be run in the host application 1102. The user script code 1106 can be written in a high-level language (e.g., C#) and a compiler 1108 that is supported by the framework 1114 can compile the user script code 1106 into an intermediate language that may be sent to a runtime environment in the framework 1114.

**[0068]** In this example 1100, a user script code transformer 1110 may be comprised in the compiler 1108, and transform the user script code 1106 into transformed debuggable user script code 1104 by injecting debuggable features into the code. For example, calls to traceback functions in the library 1112 may be injected into the user script code to allow the host application to continue functioning during debugging. Further, the host application

1102 may comprise a debugger that allows the user to debug their code, and the transformed debuggable user script code 1104 can allow the host application's UI to continue to operate during the debugging process.

**[0069]** In another embodiment, the user script code transformation component may be operably coupled to a dynamic language runtime component for the host application, which can be configured to transform the user script code during just-in-time compilation and execution of the user script code. For example, the user script code may be compiled to an intermediate code for an application development and execution framework then sent to a runtime environment within the framework. A just-in-time compilation service typically compiles intermediate language into machine language during execution of the code. In this embodiment, the just-in-time compiler can comprise the user script code transformation component to transform the user script code into transformed debuggable code during execution.

**[0070]** Still another embodiment involves a computer-readable medium comprising processor-executable instructions configured to implement one or more of the techniques presented herein. An exemplary computer-readable medium that may be devised in these ways is illustrated in Fig. 12, wherein the implementation 1200 comprises a computer-readable medium 1208 (e.g., a CD-R, DVD-R, or a platter of a hard disk drive), on which is encoded computer-readable data 1206. This computer-readable data 1206 in turn comprises a set of computer instructions 1204 configured to operate according to one or more of the principles set forth herein. In one such embodiment 1202, the processor-executable instructions 1204 may be configured to perform a method, such as the exemplary method 100 of Fig. 1, for example. In another such embodiment, the processor-executable instructions 1204 may be configured to implement a system, such as the exemplary system 1000 of Fig. 10, for example. Many such computer-readable media may be devised by those of ordinary skill in the art that are configured to operate in accordance with the techniques presented herein.

**[0071]** Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims.

**[0072]** As used in this application, the terms "component," "module," "system," "interface," and the like are generally intended to refer to a computer-related entity, either

hardware, a combination of hardware and software, software, or software in execution. For example, a component may be, but is not limited to being, a process running on a processor, a processor, an object, an executable, a thread of execution, a program, and/or a computer. By way of illustration, both an application running on a controller and the controller can be a component. One or more components may reside within a process and/or thread of execution and a component may be localized on one computer and/or distributed between two or more computers.

**[0073]** Furthermore, the claimed subject matter may be implemented as a method, apparatus, or article of manufacture using standard programming and/or engineering techniques to produce software, firmware, hardware, or any combination thereof to control a computer to implement the disclosed subject matter. The term "article of manufacture" as used herein is intended to encompass a computer program accessible from any computer-readable device, carrier, or media. Of course, those skilled in the art will recognize many modifications may be made to this configuration without departing from the scope or spirit of the claimed subject matter.

**[0074]** Fig. 10 and the following discussion provide a brief, general description of a suitable computing environment to implement embodiments of one or more of the provisions set forth herein. The operating environment of Fig. 10 is only one example of a suitable operating environment and is not intended to suggest any limitation as to the scope of use or functionality of the operating environment. Example computing devices include, but are not limited to, personal computers, server computers, hand-held or laptop devices, mobile devices (such as mobile phones, Personal Digital Assistants (PDAs), media players, and the like), multiprocessor systems, consumer electronics, mini computers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

**[0075]** Although not required, embodiments are described in the general context of "computer readable instructions" being executed by one or more computing devices. Computer readable instructions may be distributed via computer readable media (discussed below). Computer readable instructions may be implemented as program modules, such as functions, objects, Application Programming Interfaces (APIs), data structures, and the like, that perform particular tasks or implement particular abstract data types. Typically, the functionality of the computer readable instructions may be combined or distributed as desired in various environments.

**[0076]** Fig. 13 illustrates an example of a system 1310 comprising a computing device 1312 configured to implement one or more embodiments provided herein. In one configuration, computing device 1312 includes at least one processing unit 1316 and memory 1318. Depending on the exact configuration and type of computing device, memory 1318 may be volatile (such as RAM, for example), non-volatile (such as ROM, flash memory, etc., for example) or some combination of the two. This configuration is illustrated in Fig. 13 by dashed line 1314.

**[0077]** In other embodiments, device 1312 may include additional features and/or functionality. For example, device 1312 may also include additional storage (e.g., removable and/or non-removable) including, but not limited to, magnetic storage, optical storage, and the like. Such additional storage is illustrated in Fig. 13 by storage 1320. In one embodiment, computer readable instructions to implement one or more embodiments provided herein may be in storage 1320. Storage 1320 may also store other computer readable instructions to implement an operating system, an application program, and the like. Computer readable instructions may be loaded in memory 1318 for execution by processing unit 1316, for example.

**[0078]** The term "computer readable media" as used herein includes computer storage media. 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 or other data. Memory 1318 and storage 1320 are examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, 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 which can be used to store the desired information and which can be accessed by device 1312. Any such computer storage media may be part of device 1312.

**[0079]** Device 1312 may also include communication connection(s) 1326 that allows device 1312 to communicate with other devices. Communication connection(s) 1326 may include, but is not limited to, a modem, a Network Interface Card (NIC), an integrated network interface, a radio frequency transmitter/receiver, an infrared port, a USB connection, or other interfaces for connecting computing device 1312 to other computing devices. Communication connection(s) 1326 may include a wired connection or a wireless connection. Communication connection(s) 1326 may transmit and/or receive communication media.

**[0080]** The term "computer readable media" may include communication media. Communication media typically embodies computer readable instructions or other data in a "modulated data signal" such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" may include a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal.

**[0081]** Device 1312 may include input device(s) 1324 such as keyboard, mouse, pen, voice input device, touch input device, infrared cameras, video input devices, and/or any other input device. Output device(s) 1322 such as one or more displays, speakers, printers, and/or any other output device may also be included in device 1312. Input device(s) 1324 and output device(s) 1322 may be connected to device 1312 via a wired connection, wireless connection, or any combination thereof. In one embodiment, an input device or an output device from another computing device may be used as input device(s) 1324 or output device(s) 1322 for computing device 1312.

**[0082]** Components of computing device 1312 may be connected by various interconnects, such as a bus. Such interconnects may include a Peripheral Component Interconnect (PCI), such as PCI Express, a Universal Serial Bus (USB), firewire (IEEE 1394), an optical bus structure, and the like. In another embodiment, components of computing device 1312 may be interconnected by a network. For example, memory 1318 may be comprised of multiple physical memory units located in different physical locations interconnected by a network.

**[0083]** Those skilled in the art will realize that storage devices utilized to store computer readable instructions may be distributed across a network. For example, a computing device 1330 accessible via network 1328 may store computer readable instructions to implement one or more embodiments provided herein. Computing device 1312 may access computing device 1330 and download a part or all of the computer readable instructions for execution. Alternatively, computing device 1312 may download pieces of the computer readable instructions, as needed, or some instructions may be executed at computing device 1312 and some at computing device 1330.

**[0084]** Various operations of embodiments are provided herein. In one embodiment, one or more of the operations described may constitute computer readable instructions stored on one or more computer readable media, which if executed by a computing device, will cause the computing device to perform the operations described. The order in which some or all of the operations are described should not be construed as to imply that these operations are

necessarily order dependent. Alternative ordering will be appreciated by one skilled in the art having the benefit of this description. Further, it will be understood that not all operations are necessarily present in each embodiment provided herein.

**[0085]** Moreover, the word "exemplary" is used herein to mean serving as an example, instance, or illustration. Any aspect or design described herein as "exemplary" is not necessarily to be construed as advantageous over other aspects or designs. Rather, use of the word exemplary is intended to present concepts in a concrete fashion. As used in this application, the term "or" is intended to mean an inclusive "or" rather than an exclusive "or". That is, unless specified otherwise, or clear from context, "X employs A or B" is intended to mean any of the natural inclusive permutations. That is, if X employs A; X employs B; or X employs both A and B, then "X employs A or B" is satisfied under any of the foregoing instances. In addition, the articles "a" and "an" as used in this application and the appended claims may generally be construed to mean "one or more" unless specified otherwise or clear from context to be directed to a singular form.

**[0086]** Also, although the disclosure has been shown and described with respect to one or more implementations, equivalent alterations and modifications will occur to others skilled in the art based upon a reading and understanding of this specification and the annexed drawings. The disclosure includes all such modifications and alterations and is limited only by the scope of the following claims. In particular regard to the various functions performed by the above described components (e.g., elements, resources, etc.), the terms used to describe such components are intended to correspond, unless otherwise indicated, to any component which performs the specified function of the described component (*e.g.*, that is functionally equivalent), even though not structurally equivalent to the disclosed structure which performs the function in the herein illustrated exemplary implementations of the disclosure. In addition, while a particular feature of the disclosure may have been disclosed with respect to only one of several implementations, such feature may be combined with one or more other features of the other implementations as may be desired and advantageous for any given or particular application. Furthermore, to the extent that the terms "includes," "having," "has," "with," or variants thereof are used in either the detailed description or the claims, such terms are intended to be inclusive in a manner similar to the term "comprising."

What is claimed is:

1. A method (100) for enabling debuggability of user script code developed for running in a host application by transforming the user script code into debuggable code, comprising:

creating traceback methods that call back into the host application to allow the host application to cooperatively operate and update its user-interface (104);

transforming the user script code by injecting a callback, which calls to a traceback method, at one or more locations in the user script code where a stopping operation may be installed for debugging (106);

instrumenting the user script code with desired script code debugging features, comprising a callback to a traceback method (108); and

combining two or more separate script code debugging features in the user script code by using an iterator pattern transform (110).

2. The method of claim 1, transforming the user script code comprising performing language level constructs that do not utilize cooperation from an operating system's virtual machine.

3. The method of claim 1, the debugging features comprising script code that calls to a traceback method.

4. The method of claim 3, the traceback method performing one of:

detecting a debugger request for a stopping operation and stopping the debugger at that point in the user script code execution; and

detecting no debugger request for a stopping operation and treating the callback as a no-operation in the user script code.

5. The method of claim 1, debugging features comprising a user script code transform for enabling a debugging feature.

6. The method of claim 5, a user script code transform for enabling a debugging feature comprising transformed user script code that allows for creation of an allocated debugger stack frame with an attached dictionary of local variables.

7. The method of claim 5, a user script code transform for enabling a debugging feature comprising transformed user script code that allows for a function to create an allocated debugger frame object for maintaining a callstack for the debugger.

8. The method of claim 5, a user script code transform for enabling a debugging feature comprising transformed user script code that allows for a debugger to set a next active statement in a current frame without executing user code between a current statement and

the next statement by injecting one or more switch tables at a location of a set-next statement in a function.

9. The method of claim 5, instrumenting the user script code with desired script code debugging features comprising:

compiling a function into a separate header and body;

storing the function body in a global table;

allowing a stored function body to be updated separately from its corresponding header; and

allowing the function header to fetch an updated function body from the global table.

10. The method of claim 5, a user script code transform for enabling a debugging feature comprising transforming the user script code by inserting a call to a traceback in a synchronous exception.

11. The method of claim 1, combining two or more separate script code debugging features in the user script code by using iterator pattern transform comprising:

inserting script code that maps function headers to a global table of function bodies;

inserting script code that has respective function headers, having debugging features to be combined, call an iterator pattern function; and

retrieving corresponding updated function bodies that have been transformed to include code enabling a debugging feature using the iterator pattern function.

12. A system (1000) for transforming user script code, developed for running in a host application, into debuggable code, comprising:

traceback methods (1008) configured to call back into the host application to allow the host application to cooperatively operate and update its user-interface;

a user script code transformation component (1002) configured to transform user script code developed for running in a host application, comprising:

a traceback method callback instrumentation component (1004) configured to inject a callback, which calls to a traceback method, at one or more locations in the user script code where a stopping operation may be installed for debugging; and

a user script code debugging feature combiner (1006) configured to combine two or more script code debugging features in the user script code by using an iterator pattern transform.

13. The system of claim 12, the user script code transformation component operably coupled to a dynamic language runtime component for the host application, configured to

transform the user script code during just-in-time compilation and execution of the user script code.

14. The system of claim 12, the user script code transformation component comprising a function separation component configured to:

compile a function into a separate header and body; and

store the function body in a global table comprising function bodies mapped to corresponding function headers.

15. The system of claim 14, the user script code debugging feature combiner configured to:

insert script code that maps function headers to a global table of function bodies;

insert script code that has respective function headers, having debugging features to be combined, call an iterator pattern function; and

retrieve corresponding updated function bodies that have been transformed to include code enabling a debugging feature using the iterator pattern function.

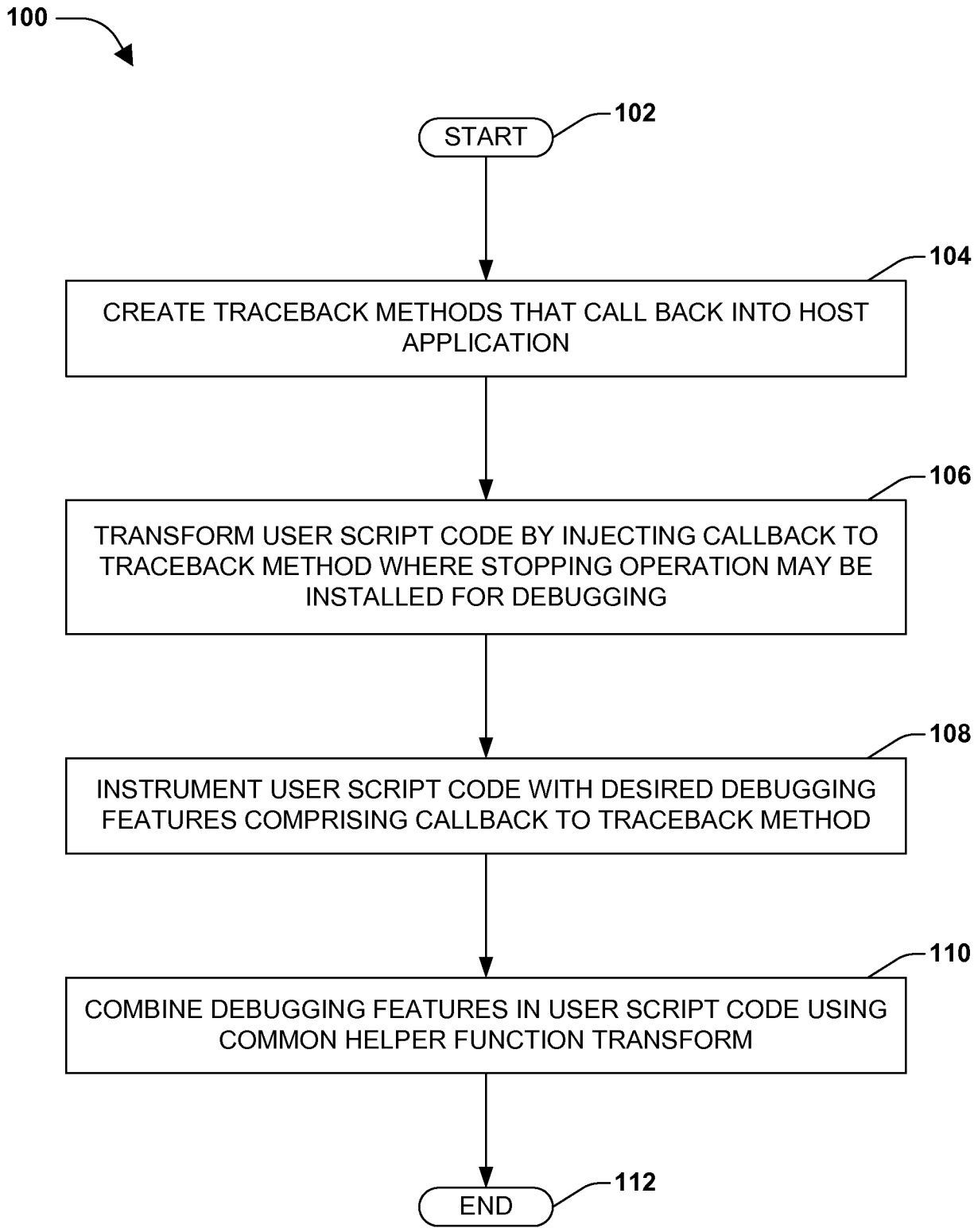
100
START
102
CREATE TRACEBACK METHODS THAT CALL BACK INTO HOST APPLICATION
104
TRANSFORM USER SCRIPT CODE BY INJECTING CALLBACK TO TRACEBACK METHOD WHERE STOPPING OPERATION MAY BE INSTALLED FOR DEBUGGING
106
INSTRUMENT USER SCRIPT CODE WITH DESIRED DEBUGGING FEATURES COMPRISING CALLBACK TO TRACEBACK METHOD
108
COMBINE DEBUGGING FEATURES IN USER SCRIPT CODE USING COMMON HELPER FUNCTION TRANSFORM
110
END
112

FIG. 1

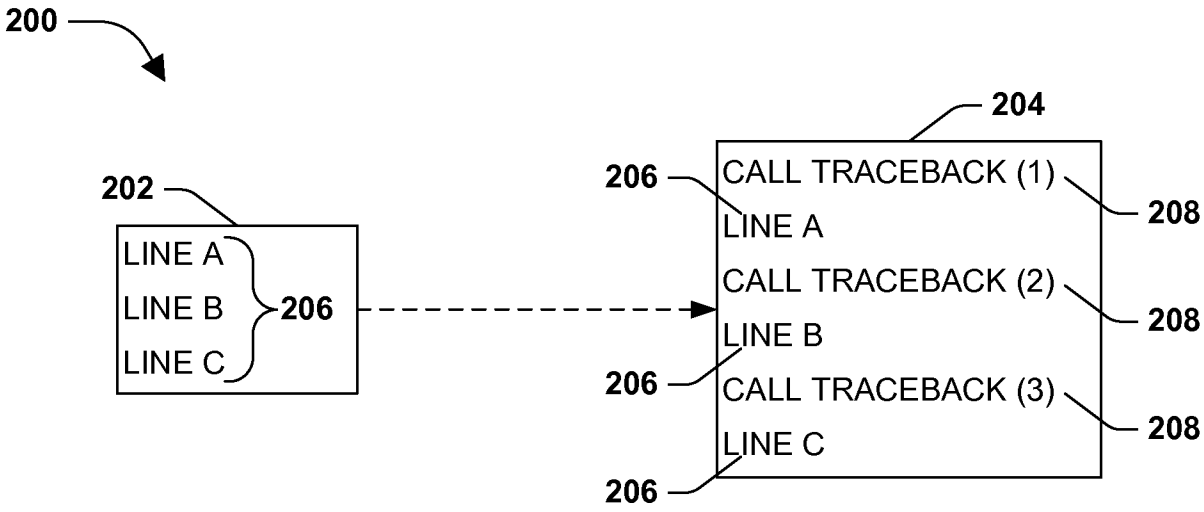
200
202
LINE A
LINE B
LINE C
206
204
206
CALL TRACEBACK (1)
LINE A
CALL TRACEBACK (2)
LINE B
CALL TRACEBACK (3)
LINE C
208
208
208
206
206

FIG. 2

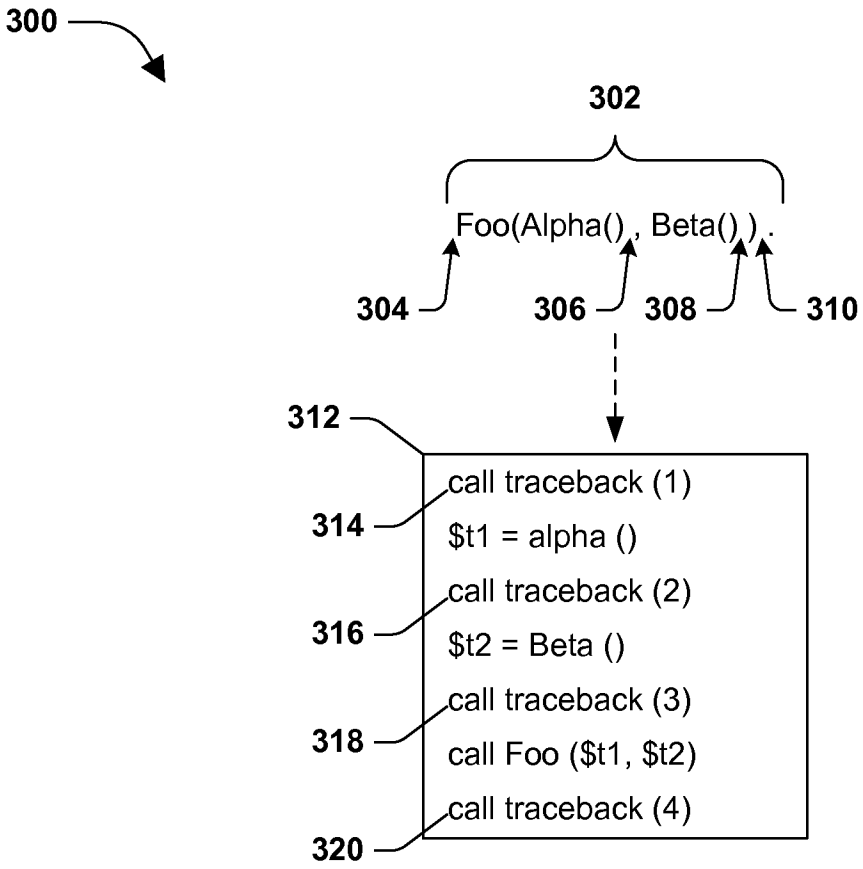
300
302
Foo(Alpha(), Beta() ).
304
306
308
310
312
call traceback (1)
$t1 = alpha ()
call traceback (2)
$t2 = Beta ()
call traceback (3)
call Foo ($t1, $t2)
call traceback (4)
314
316
318
320

FIG. 3

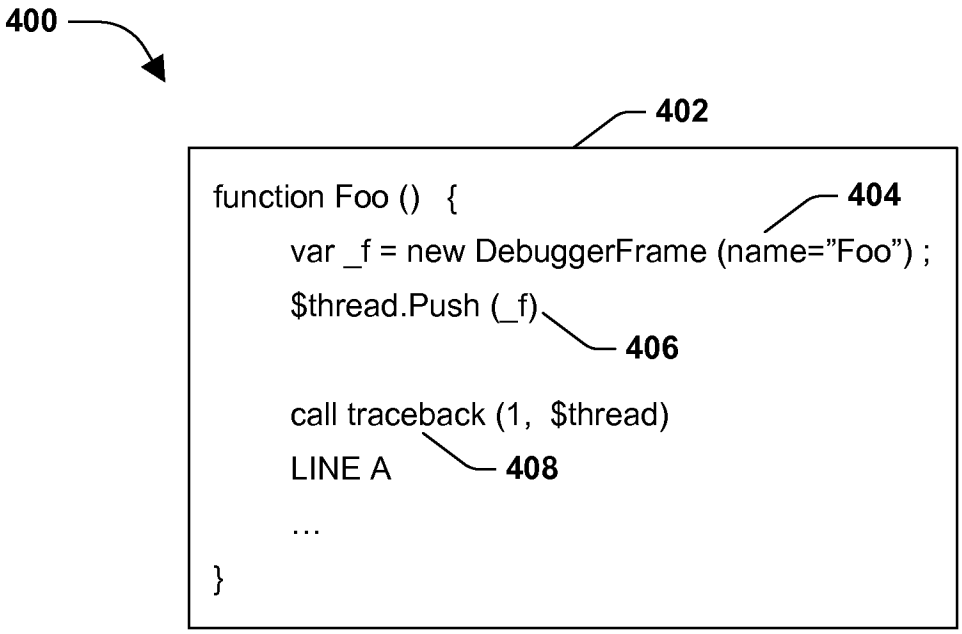
400
402
function Foo () {
404
var _f = new DebuggerFrame (name="Foo") ;
$thread.Push (_f)
406
call traceback (1, $thread)
LINE A
408
…
}

FIG. 4

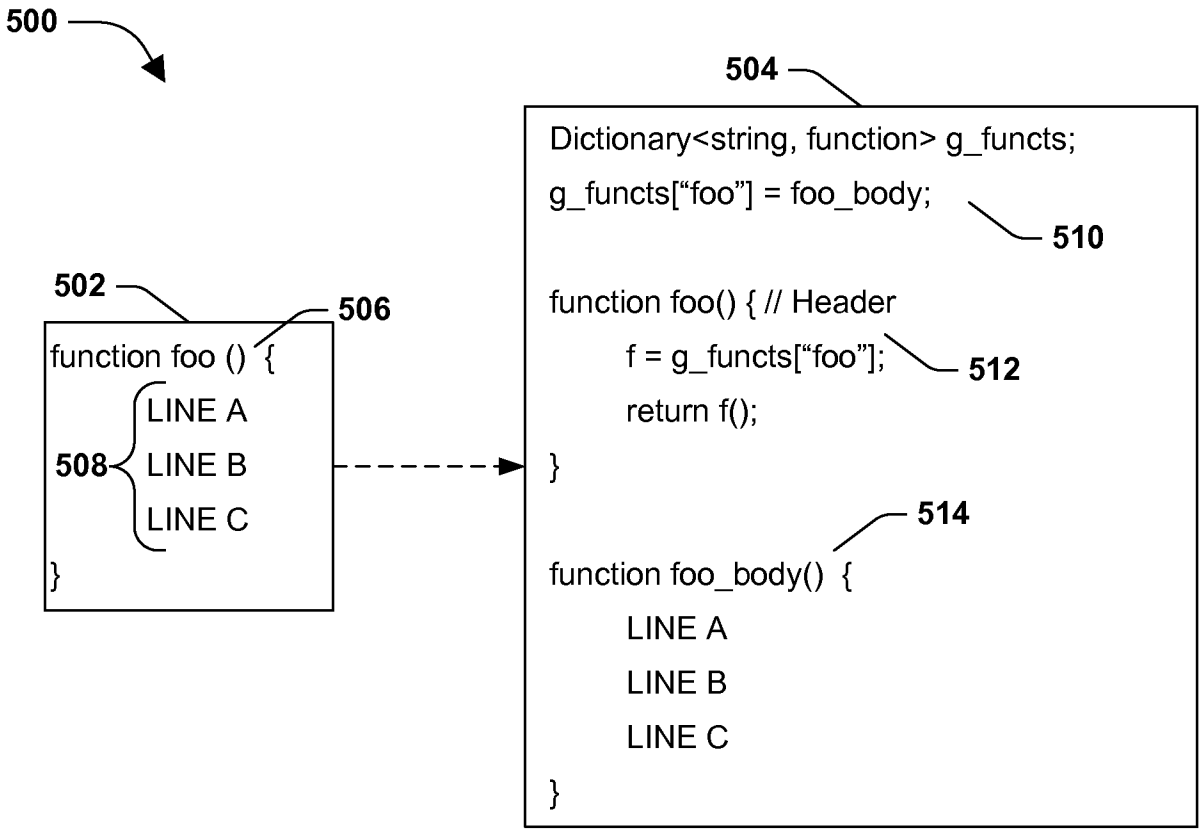
500
504
Dictionary<string, function> g_functs;
g_functs["foo"] = foo_body;
510
function foo() { // Header
f = g_functs["foo"];
512
return f();
}
514
function foo_body() {
LINE A
LINE B
LINE C
}
502
506
function foo () {
508
LINE A
LINE B
LINE C
}

FIG. 5

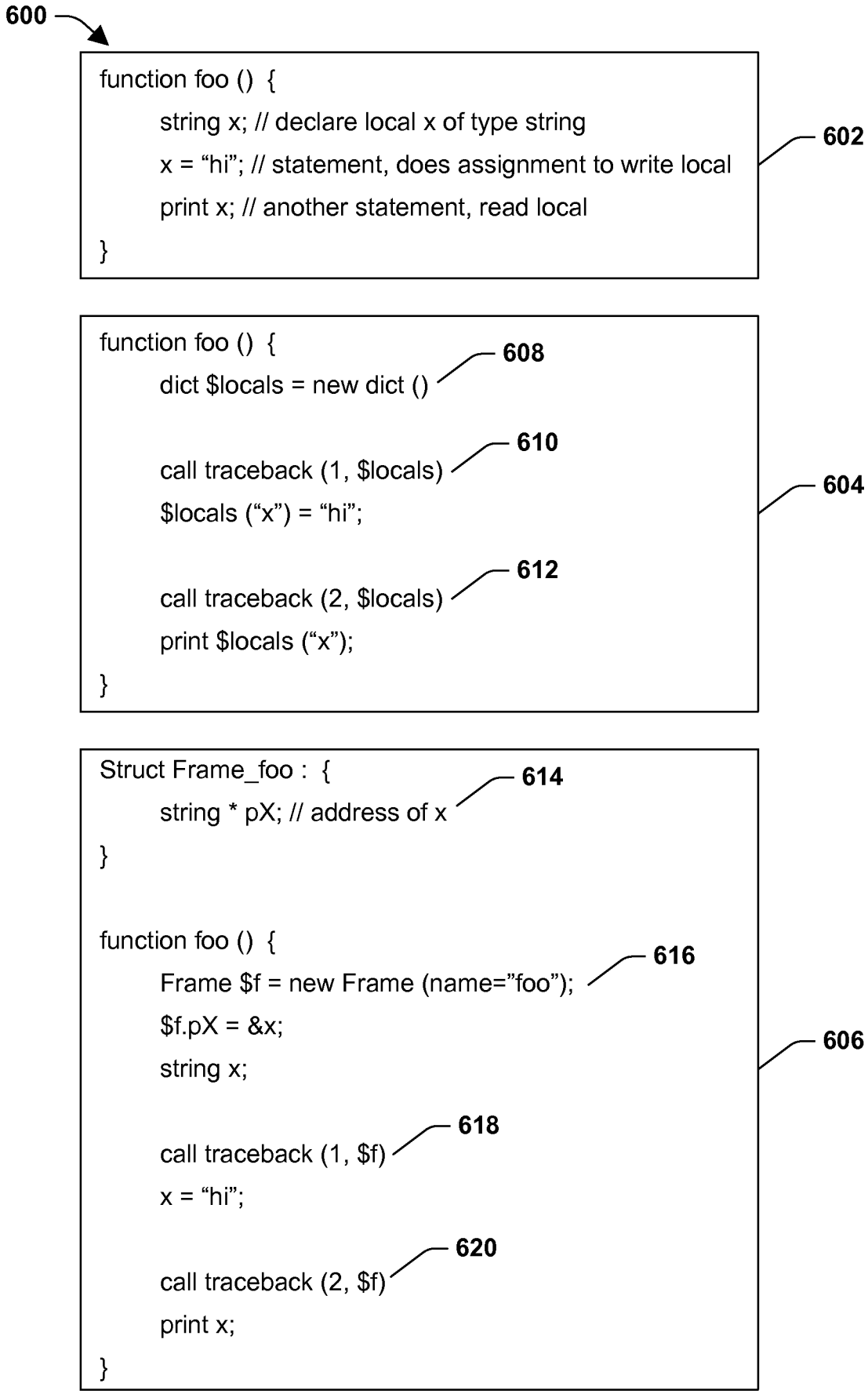
600
function foo () {
string x; // declare local x of type string
x = "hi"; // statement, does assignment to write local
print x; // another statement, read local
}
602
function foo () {
dict $locals = new dict ()
608
call traceback (1, $locals)
610
$locals ("x") = "hi";
call traceback (2, $locals)
612
print $locals ("x");
}
604
Struct Frame_foo : {
string * pX; // address of x
614
}
function foo () {
Frame $f = new Frame (name="foo");
616
$f.pX = &x;
string x;
call traceback (1, $f)
618
x = "hi";
call traceback (2, $f)
620
print x;
}
606

FIG. 6

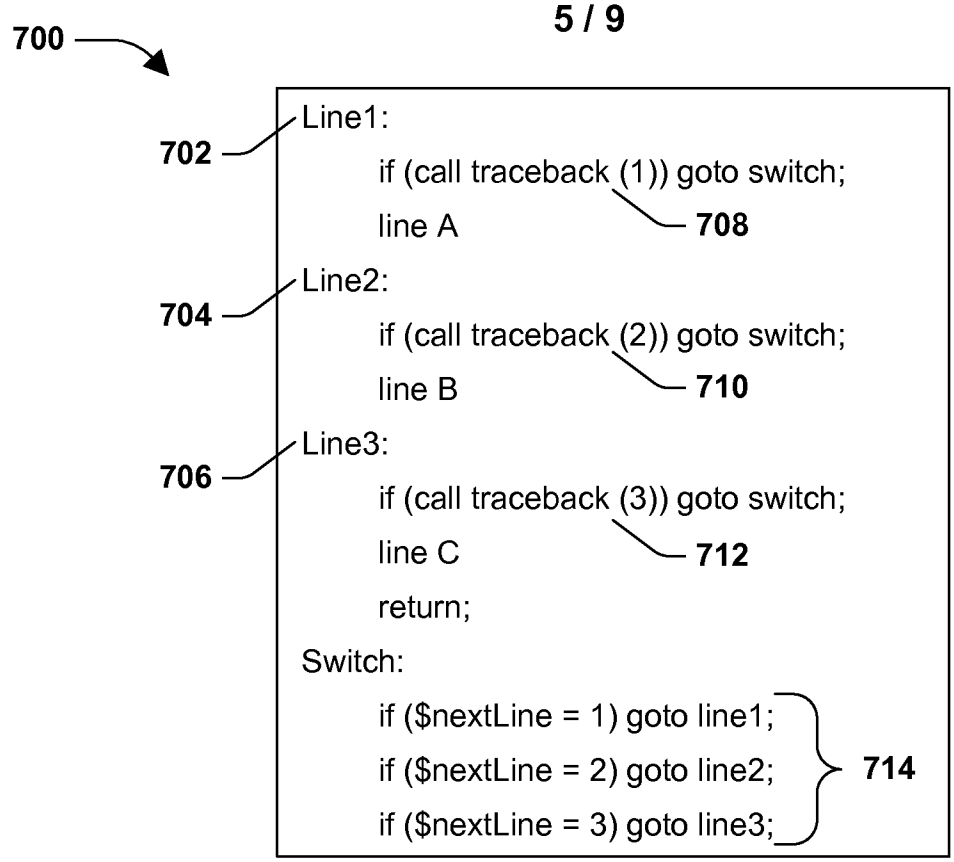
700
702
Line1:
if (call traceback (1)) goto switch;
line A
708
704
Line2:
if (call traceback (2)) goto switch;
line B
710
706
Line3:
if (call traceback (3)) goto switch;
line C
712
return;
Switch:
if ($nextLine = 1) goto line1;
if ($nextLine = 2) goto line2;
if ($nextLine = 3) goto line3;
714

FIG. 7

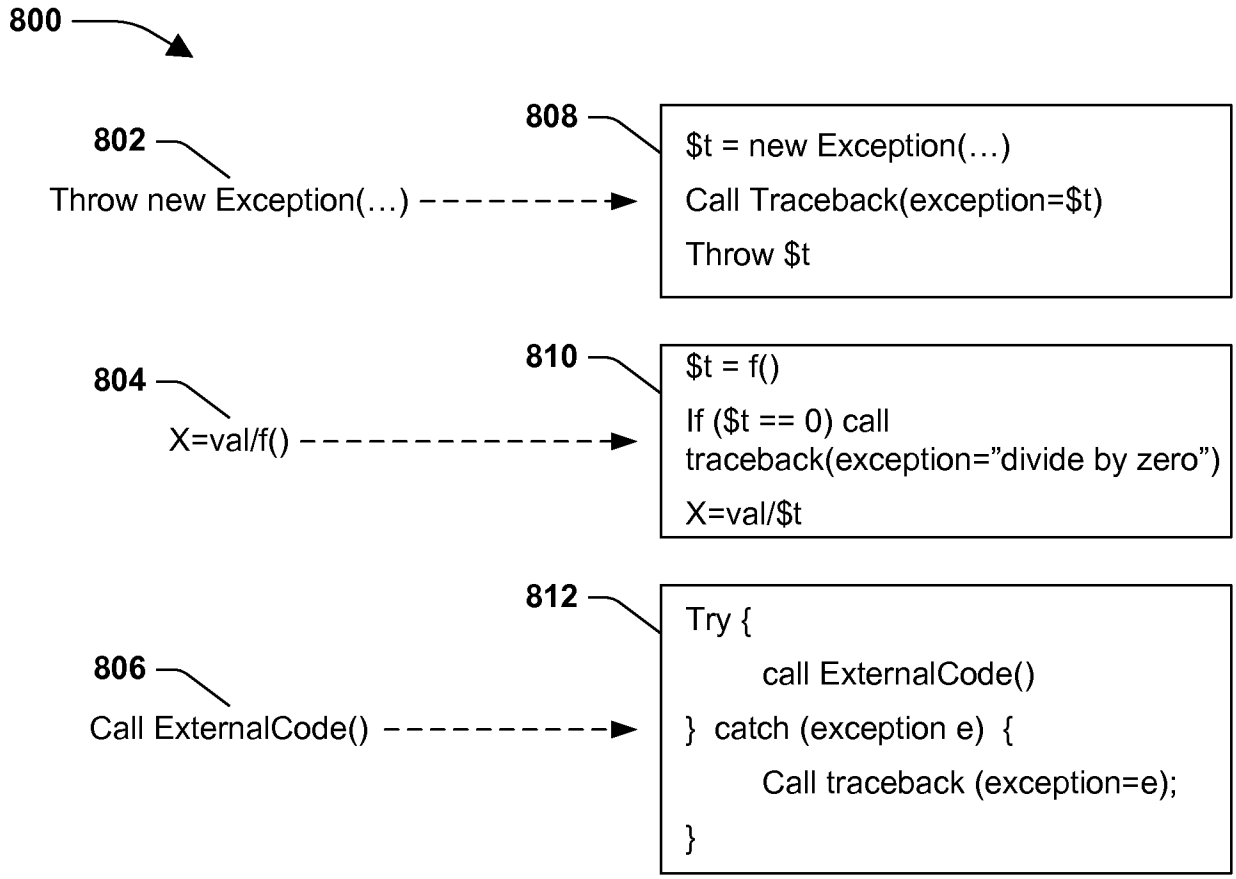
800
802
Throw new Exception(…)
808
$t = new Exception(…)
Call Traceback(exception=$t)
Throw $t
804
X=val/f()
810
$t = f()
If ($t == 0) call traceback(exception="divide by zero")
X=val/$t
806
Call ExternalCode()
812
Try {
call ExternalCode()
} catch (exception e) {
Call traceback (exception=e);
}

FIG. 8

900

```
Dictionary<string, function> g_functs;                    902
g_functs["foo"] = foo_body;

// shared common helper for all function headers          904
function CommonHelper(string name) {
   IEnumerable e = g_functs[name](); // gets latest version of foo
   $thread.push(new DebuggerStackFrame(name, e));
   Try {                                                  906
     e.MoveNext(); // move to first traceback spot

     While(!e.IsAtEnd()) {                                908
       Call traceback(e)
       If (SetNextStatementRequested()) {
         Update(e, $targetLine);
         Continue; // loop back to traceback at the target line
       }

       Try {
         e.MoveNext(); // executes function body up to the next point
       } catch (Exception ex) {
         Call traceback(exception=ex)
       }
     }
   } finally {
     $thread.Pop()
   }
}

                                                          910
function foo() { // Header
    CommonHelper ("foo")
}
                                                          912
// foo's body is converted into an iterator. The header drives the iterator.
// each yield point becomes a traceback.
function foo_body() {
   yield 1
   LINE A
   yield 2
   LINE B
   yield 3
   LINE C
}
```

FIG. 9

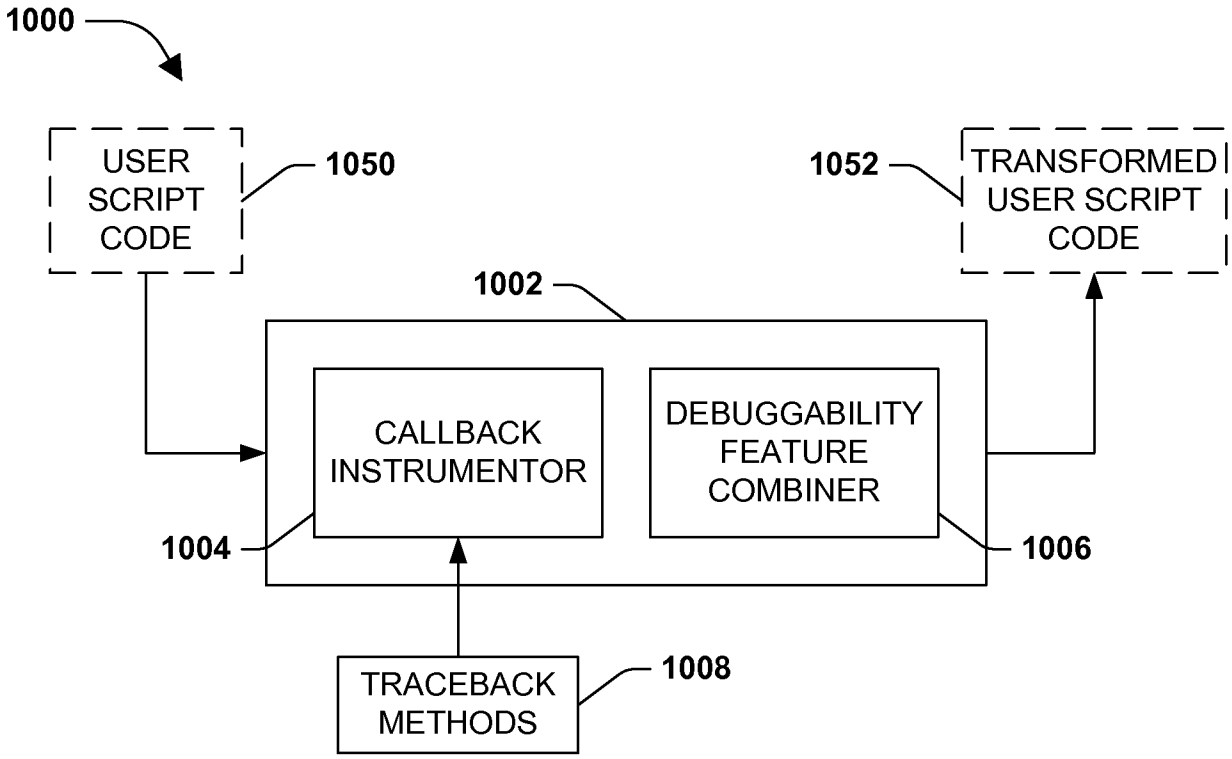
1000
USER SCRIPT CODE
1050
1052
TRANSFORMED USER SCRIPT CODE
1002
CALLBACK INSTRUMENTOR
DEBUGGABILITY FEATURE COMBINER
1004
1006
TRACEBACK METHODS
1008

FIG. 10

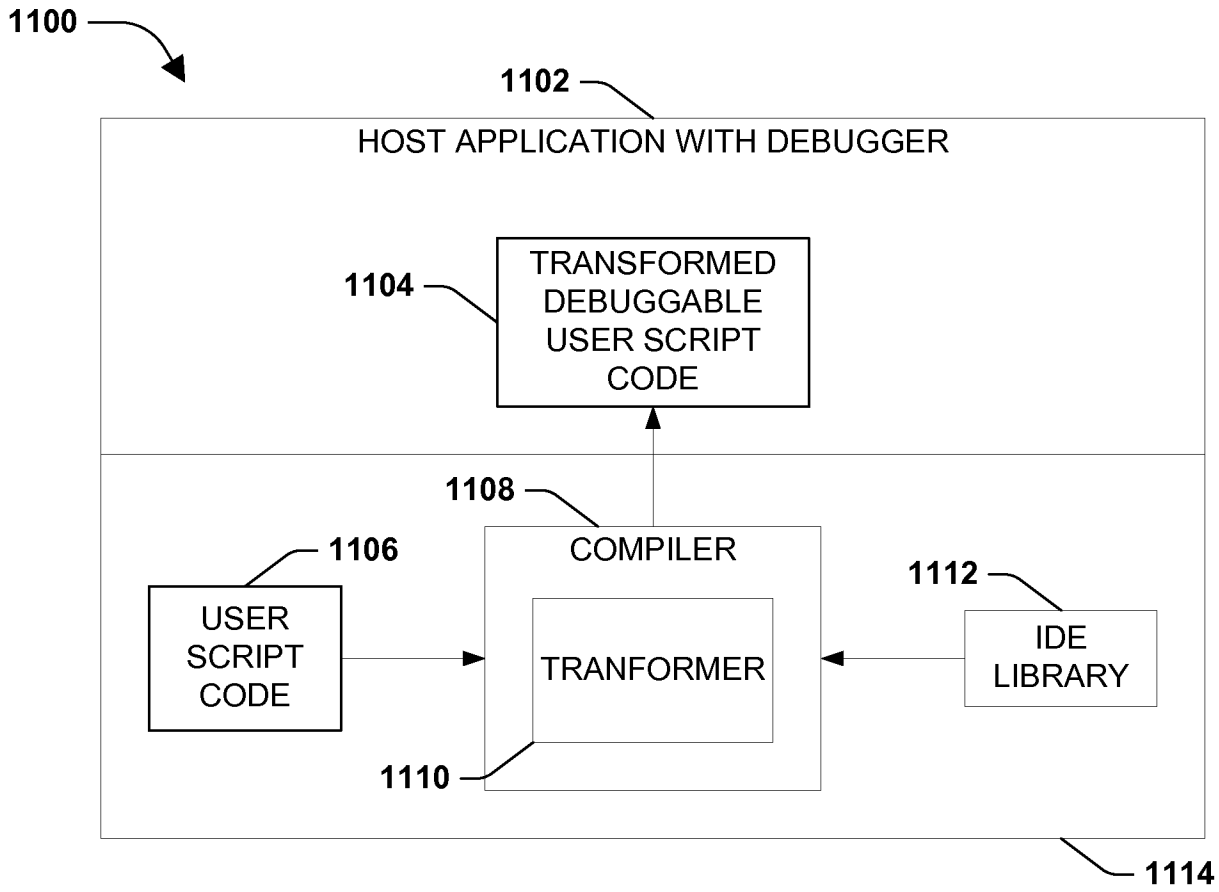
1100
1102
HOST APPLICATION WITH DEBUGGER
1104
TRANSFORMED DEBUGGABLE USER SCRIPT CODE
1108
COMPILER
1106
USER SCRIPT CODE
TRANFORMER
1112
IDE LIBRARY
1110
1114

FIG. 11

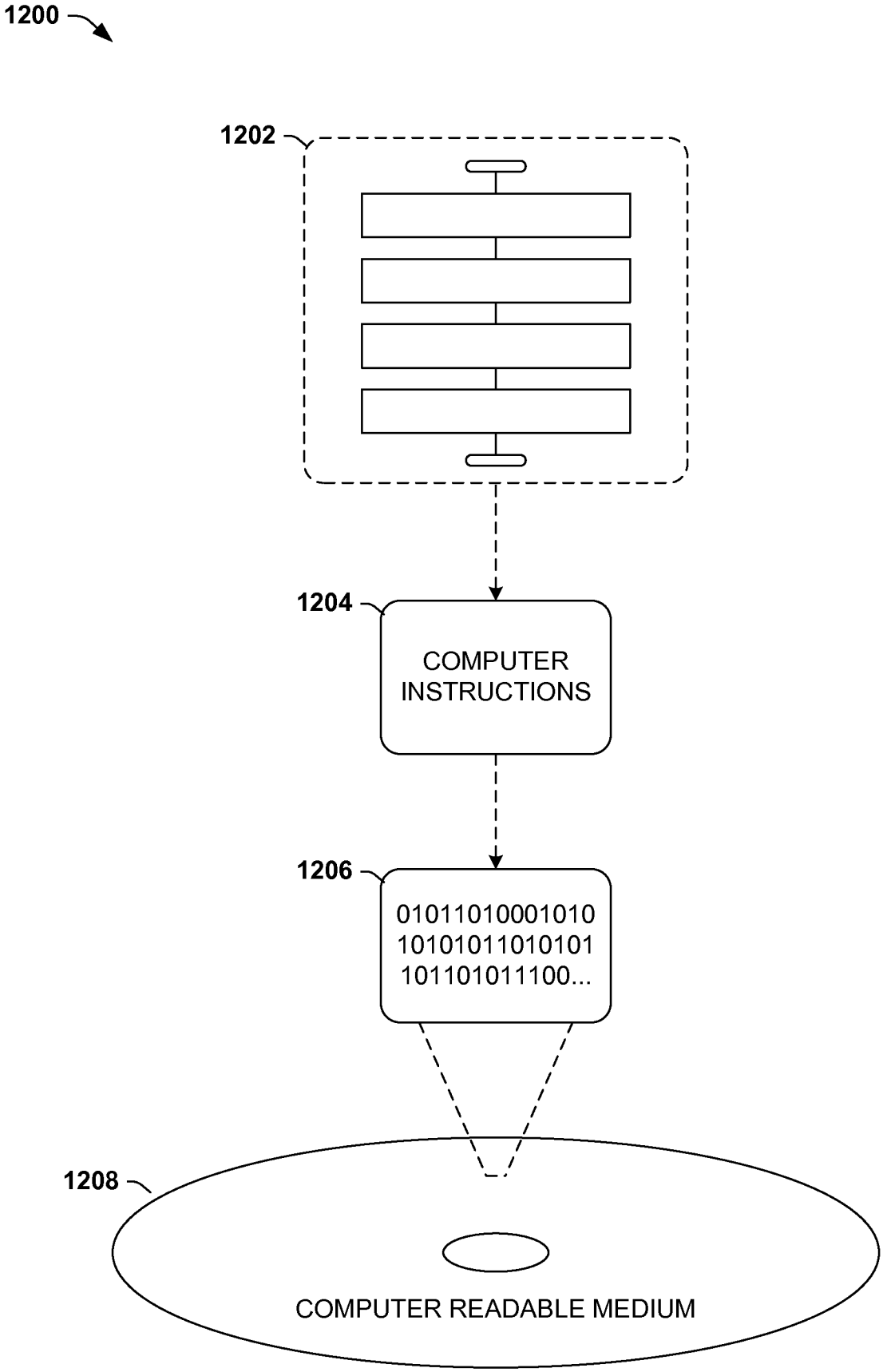
1200
1202
1204
COMPUTER INSTRUCTIONS
1206
01011010001010
10101011010101
101101011100...
1208
COMPUTER READABLE MEDIUM

FIG. 12

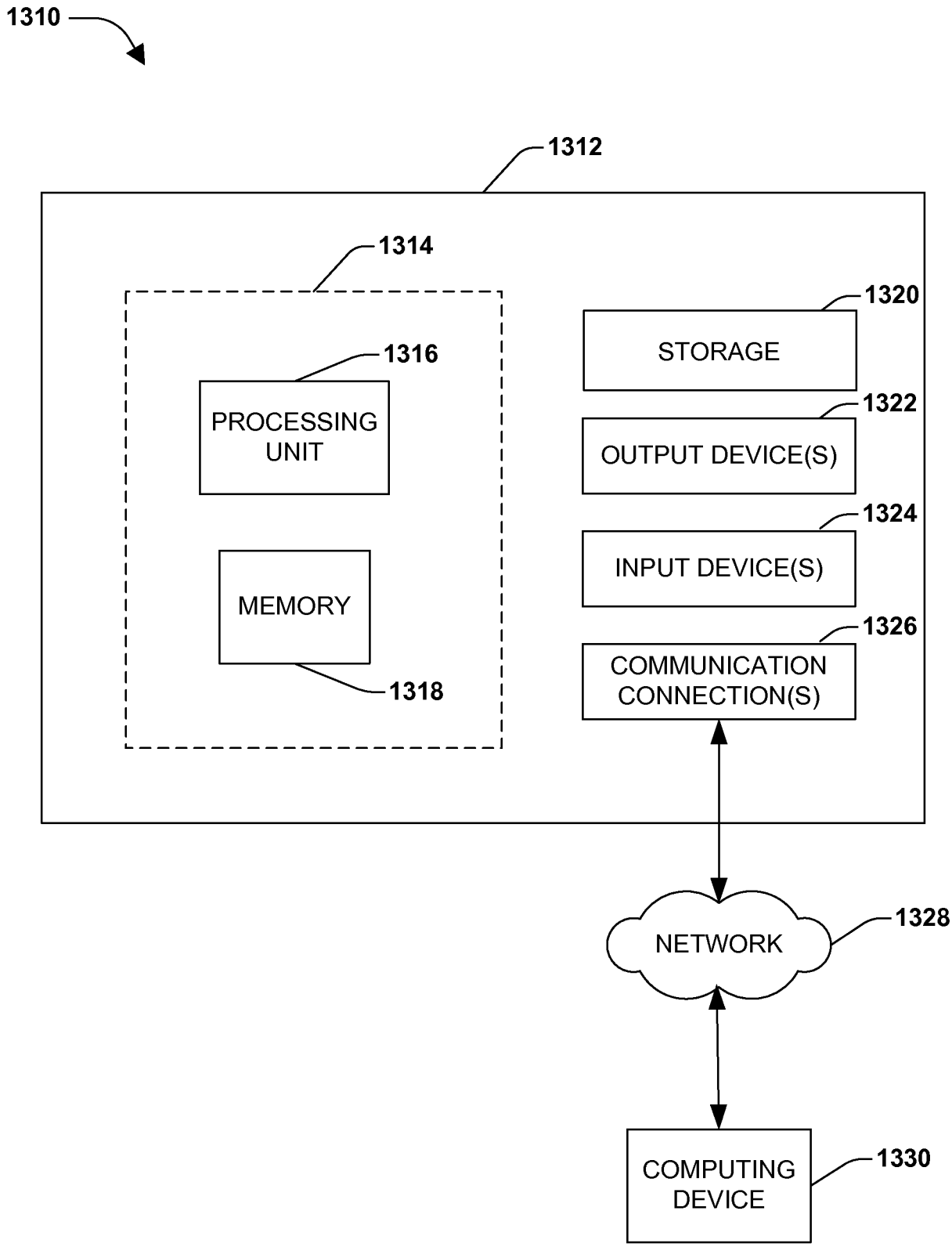
1310
1312
1314
1316
PROCESSING UNIT
MEMORY
1318
1320
STORAGE
1322
OUTPUT DEVICE(S)
1324
INPUT DEVICE(S)
1326
COMMUNICATION CONNECTION(S)
NETWORK
1328
COMPUTING DEVICE
1330

FIG. 13