Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2687672 A1 2011/06/07

(21) 2 687 672

(12) DEMANDE DE BREVET CANADIEN
CANADIAN PATENT APPLICATION

(13) A1

(22) Date de dépôt/Filing Date: 2009/12/07

(41) Mise à la disp. pub./Open to Public Insp.: 2011/06/07

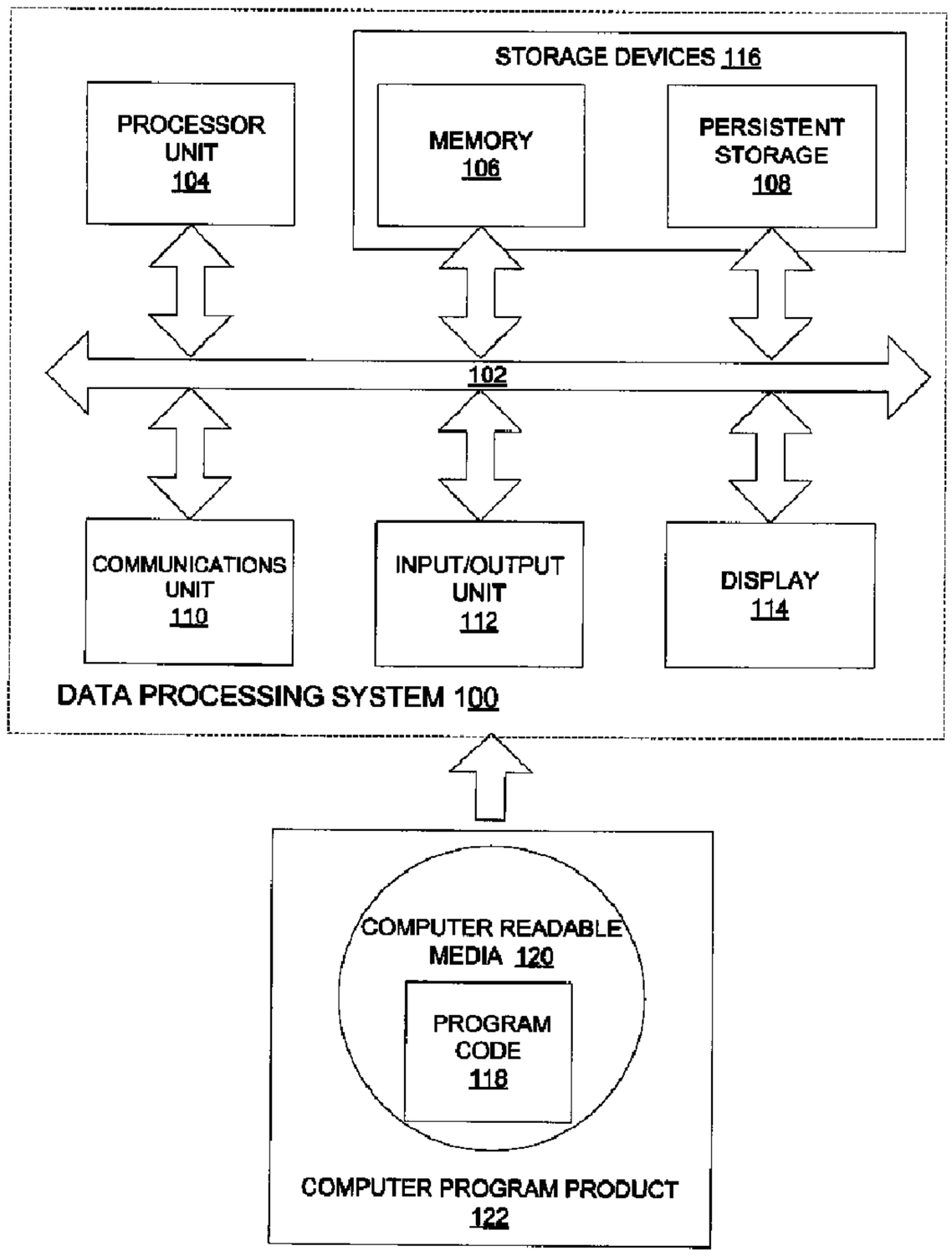
(51) Cl.Int./Int.Cl. *G06F 9/44* (2006.01),
G06F 11/36 (2006.01)

(71) Demandeur/Applicant:
IBM CANADA LIMITED - IBM CANADA LIMITEE, CA

(72) Inventeurs/Inventors:
MOSTAFA, MOHAMMED, CA;
TESSIER, JOSHUA PETER, CA;
HILLIS, LINDA, CA

(74) Agent: WANG, PETER

(54) Titre : GENERATION INTELLIGENTE DE GRAPHE D'APPEL
(54) Title: INTELLIGENT CALL GRAPH GENERATION



(57) Abrégé/Abstract:
An illustrative embodiment of a computer-implemented process for intelligent call graph generation receives a request to build a call graph to form a received request, receives a source code associated with the received request, receives configuration options, and

CA 2687672 A1 2011/06/07

(21) 2 687 672

(13) A1

(57) Abrégé(suite)/Abstract(continued):
determines whether an entry point for the call graph is identified in the source code. Responsive to a determination that the entry point for the call graph is identified, analyzes dependencies in the source code, and identifies exclusions in the source code using the analyzed dependencies to form identified exclusions. The computer-implemented process determines whether analysis required to build a compact call graph is complete and responsive to a determination that analysis required to build a compact call graph is complete, generates the compact call graph without the identified exclusions.

ABSTRACT OF THE DISCLOSURE

An illustrative embodiment of a computer-implemented process for intelligent call graph generation receives a request to build a call graph to form a received request, receives a source code associated with the received request, receives configuration options, and determines whether an entry point for the call graph is identified in the source code. Responsive to a determination that the entry point for the call graph is identified, analyzes dependencies in the source code, and identifies exclusions in the source code using the analyzed dependencies to form identified exclusions. The computer-implemented process determines whether analysis required to build a compact call graph is complete and responsive to a determination that analysis required to build a compact call graph is complete, generates the compact call graph without the identified exclusions.

INTELLIGENT CALL GRAPH GENERATION

BACKGROUND

1. Technical Field:

[0001] This disclosure relates generally to generating call graphs in a data processing system and more specifically to intelligent call graph generation within a data processing system.

2. Description of the Related Art:

[0002] Data flow analysis is a form of static analysis used to analyze source code in depth and identify potential problems. Data flow analysis is a technique often used to identify problems including the use of NULL pointers, buffer overflows and other serious coding mistakes.

[0003] To accomplish this task, data flow analysis leverages a data flow engine which constructs a call graph. The call graph is a data structure typically containing a very large set of nodes that represent function calls and variables of a program or application source code being evaluated. The generated call graph, which is augmented with the data flow information, can be used to track the definition of variables and identify where the variables are used incorrectly.

[0004] Unfortunately, the task of creating a useable call graph is nontrivial. Although generating a call graph is simple in nature, handling the graph is not. The graph can contain an unlimited number of nodes and may consume more memory on the system than available. Therefore, using a call graph is often impractical despite a wealth of information provided by the call graph.

BRIEF SUMMARY

[0005] According to one embodiment, a computer-implemented process for intelligent call graph generation receives a request to build a call graph to form a received request, receives a source code associated with the received request, receives configuration options, and determines whether an entry point for the call graph is identified in the source code. Responsive to a determination that the entry point for the call graph is identified, analyzes dependencies in the source code, and identifies exclusions in the source code using the analyzed dependencies to form identified exclusions, determines whether analysis required to build a compact call graph is complete, and responsive to a determination that analysis required to build a compact call graph is complete, generates the compact call graph without the identified exclusions.

[0006] According to another embodiment, a computer program product for intelligent call graph generation comprises a computer recordable-type media containing computer executable program code stored thereon. The computer executable program code comprises computer executable program code for receiving a request to build a call graph to form a received request, computer executable program code for receiving a source code associated with the received request, computer executable program code for receiving configuration options, computer executable program code for determining whether an entry point for the call graph is identified in the source code, computer executable program code responsive to a determination that the entry point for the call graph is identified, for analyzing dependencies in the source code, computer executable program code for identifying exclusions in the source code using the analyzed dependencies to form identified exclusions, computer executable program code for determining whether analysis required to build a compact call graph is complete, and computer executable program code responsive to a determination that analysis required to build a compact call graph is complete, for generating the compact call graph without the identified exclusions.

[0007] According to another embodiment, an apparatus for intelligent call graph generation, the apparatus comprising a communications fabric, a memory connected to the communications fabric, wherein the memory contains computer executable program code, a communications unit connected to the communications fabric, an input/output unit connected to the communications

fabric, a display connected to the communications fabric, and a processor unit connected to the communications fabric. The processor unit executes the computer executable program code to direct the apparatus to receive a request to build a call graph to form a received request, receive a source code associated with the received request, receive configuration options, determine whether an entry point for the call graph is identified in the source code, responsive to a determination that the entry point for the call graph is identified, analyze dependencies in the source code, identify exclusions in the source code using the analyzed dependencies to form identified exclusions, determines whether analysis required to build a compact call graph is complete, and responsive to a determination that analysis required to build a compact call graph is complete, generates the compact call graph without the identified exclusions.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

[0008] For a more complete understanding of this disclosure, reference is now made to the following brief description, taken in conjunction with the accompanying drawings and detailed description, wherein like reference numerals represent like parts.

[0009] Figure 1 is a block diagram of an exemplary data processing system operable for various embodiments of the disclosure;

[0010] Figure 2 is a block diagram of a compilation system in accordance with various embodiments of the disclosure;

[0011] Figure 3 is a block diagram of a call graph output from the compilation system of **Figure 2** in accordance with one embodiment of the disclosure; and

[0012] Figure 4 is a flowchart of a call graph generating process using the compilation system of **Figure 2**, in accordance with one embodiment of the disclosure.

DETAILED DESCRIPTION

[0013] Although an illustrative implementation of one or more embodiments is provided below, the disclosed systems and/or methods may be implemented using any number of techniques. This disclosure should in no way be limited to the illustrative implementations, drawings, and techniques illustrated below, including the exemplary designs and implementations illustrated and described herein, but may be modified within the scope of the appended claims along with their full scope of equivalents.

[0014] As will be appreciated by one skilled in the art, the present disclosure may be embodied as a system, method or computer program product. Accordingly, the present disclosure may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, etc.) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a "circuit," "module," or "system." Furthermore, the present invention may take the form of a computer program product tangibly embodied in any medium of expression with computer usable program code embodied in the medium.

[0015] Computer program code for carrying out operations of the present disclosure may be written in any combination of one or more programming languages, including an object oriented programming language such as Java™, Smalltalk, C++, or the like and conventional procedural programming languages, such as the "C" programming language or similar programming languages. Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc., in the United States, other countries or both. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

[0016] The present disclosure is described below with reference to flowchart illustrations and/or block diagrams of methods, apparatus, systems, and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations

and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer program instructions.

[0017] These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks. These computer program instructions may also be stored in a computer readable medium that can direct a computer or other programmable data processing apparatus to function in a particular manner, such that the instructions stored in the computer readable medium produce an article of manufacture including instruction means which implement the function/act specified in the flowchart and/or block diagram block or blocks.

[0018] The computer program instructions may also be loaded onto a computer or other programmable data processing apparatus to cause a series of operational steps to be performed on the computer or other programmable apparatus to produce a computer-implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0019] Turning now to **Figure 1** a block diagram of an exemplary data processing system operable for various embodiments of the disclosure is presented. In this illustrative example, data processing system **100** includes communications fabric **102**, which provides communications between processor unit **104**, memory **106**, persistent storage **108**, communications unit **110**, input/output (I/O) unit **112**, and display **114**.

[0020] Processor unit **104** serves to execute instructions for software that may be loaded into memory **106**. Processor unit **104** may be a set of one or more processors or may be a multi-processor core, depending on the particular implementation. Further, processor unit **104** may be implemented using one or more heterogeneous processor systems in which a main processor is

present with secondary processors on a single chip. As another illustrative example, processor unit **104** may be a symmetric multi-processor system containing multiple processors of the same type.

[0021] Memory **106** and persistent storage **108** are examples of storage devices **116**. A storage device is any piece of hardware that is capable of storing information, such as, for example without limitation, data, program code in functional form, and/or other suitable information either on a temporary basis and/or a permanent basis. Memory **106**, in these examples, may be, for example, a random access memory or any other suitable volatile or non-volatile storage device. Persistent storage **108** may take various forms depending on the particular implementation. For example, persistent storage **108** may contain one or more components or devices. For example, persistent storage **108** may be a hard drive, a flash memory, a rewritable optical disk, a rewritable magnetic tape, or some combination of the above. The media used by persistent storage **108** also may be removable. For example, a removable hard drive may be used for persistent storage **108**.

[0022] Communications unit **110,** in these examples, provides for communications with other data processing systems or devices. In these examples, communications unit **110** is a network interface card. Communications unit **110** may provide communications through the use of either or both physical and wireless communications links.

[0023] Input/output unit **112** allows for input and output of data with other devices that may be connected to data processing system **100**. For example, input/output unit **112** may provide a connection for user input through a keyboard, a mouse, and/or some other suitable input device. Further, input/output unit **112** may send output to a printer. Display **114** provides a mechanism to display information to a user.

[0024] Instructions for the operating system, applications and/or programs may be located in storage devices **116,** which are in communication with processor unit **104** through communications fabric **102.** In these illustrative examples the instructions are in a functional form on persistent storage **108**. These instructions may be loaded into memory **106** for execution by processor unit **104**. The processes of the different embodiments may be performed by

processor unit **104** using computer-implemented instructions, which may be located in a memory, such as memory **106**.

[0025] These instructions are referred to as program code, computer usable program code, or computer readable program code that may be read and executed by a processor in processor unit **104**. The program code in the different embodiments may be embodied on different physical or tangible computer readable media, such as memory **106** or persistent storage **108**.

[0026] Program code **118** is located in a functional form on computer readable media **120** that is selectively removable and may be loaded onto or transferred to data processing system **100** for execution by processor unit **104**. Program code **118** and computer readable media **120** form computer program product **122** in these examples. In one example, computer readable media **120** may be in a tangible form, such as, for example, an optical or magnetic disc that is inserted or placed into a drive or other device that is part of persistent storage **108** for transfer onto a storage device, such as a hard drive that is part of persistent storage **108**. In a tangible form, computer readable media **120** also may take the form of a persistent storage, such as a hard drive, a thumb drive, or a flash memory that is connected to data processing system **100**. The tangible form of computer readable media **120** is also referred to as computer recordable storage media. In some instances, computer readable media **120** may not be removable.

[0027] Alternatively, program code **118** may be transferred to data processing system **100** from computer readable media **120** through a communications link to communications unit **110** and/or through a connection to input/output unit **112**. The communications link and/or the connection may be physical or wireless in the illustrative examples. The computer readable media also may take the form of non-tangible media, such as communications links or wireless transmissions containing the program code.

[0028] In some illustrative embodiments, program code **118** may be downloaded over a network to persistent storage **108** from another device or data processing system for use within data processing system **100**. For instance, program code stored in a computer readable storage medium in a server data processing system may be downloaded over a network from the server

to data processing system **100**. The data processing system providing program code **118** may be a server computer, a client computer, or some other device capable of storing and transmitting program code **118.**

[0029] The different components illustrated for data processing system **100** are not meant to provide architectural limitations to the manner in which different embodiments may be implemented. The different illustrative embodiments may be implemented in a data processing system including components in addition to or in place of those illustrated for data processing system **100**. Other components shown in **Figure 1** can be varied from the illustrative examples shown. The different embodiments may be implemented using any hardware device or system capable of executing program code. As one example, the data processing system may include organic components integrated with inorganic components and/or may be comprised entirely of organic components excluding a human being. For example, a storage device may be comprised of an organic semiconductor.

[0030] As another example, a storage device in data processing system **100** may be any hardware apparatus that may store data. Memory **106**, persistent storage **108** and computer readable media **120** are examples of storage devices in a tangible form.

[0031] In another example, a bus system may be used to implement communications fabric **102** and may be comprised of one or more buses, such as a system bus or an input/output bus. Of course, the bus system may be implemented using any suitable type of architecture that provides for a transfer of data between different components or devices attached to the bus system. Additionally, a communications unit may include one or more devices used to transmit and receive data, such as a modem or a network adapter. Further, a memory may be, for example, memory **106** or a cache such as found in an interface and memory controller hub that may be present in communications fabric **102**.

[0032] Using data processing system **100** of **Figure 1** as an example, an illustrative embodiment provides the computer-implemented process stored in memory **106**, executed by processor unit **104**, for intelligent call graph generation. Processor unit **104** receives a request through

communications unit **110** or input/output unit **112** to build a call graph to form a received request, and receives a source code associated with the received request, and configuration options, from storage devices **116**. Processor unit **104** determines whether an entry point for the call graph is identified in the source code. Responsive to a determination that the entry point for the call graph is identified, processor unit **104** analyzes dependencies in the source code, identifies exclusions in the source code using the analyzed dependencies to form identified exclusions, and responsive to a determination that analysis required to build a compact call graph is complete, generates the compact call graph without the identified exclusions. The generated compact call graph is returned to the requester through communications unit **110** or stored in storage devices **116**.

[0033] The adding of an *intelligent pruning* technique to the static analysis engine typically avoids creating large call graphs by eliminating unnecessary elements before the elements are built into the call graph. When using a call graph in a static analysis context, a search is typically initiated for a set of defined method calls or function calls within the call graph. Rather than create the call graph without bounds, the call graph can be created where a built in pruning phase can typically intelligently reduce the overall size of the graph. Unnecessary branches are eliminated from the call graph which typically results in reduced traversal times (the time it takes to analyze a call graph) and in memory consumption (the overall size of the graph), thereby producing a compact call graph as compared to a call graph generated using a traditional process.

[0034] In another example, a computer-implemented process, using program code **118** stored in memory **106** or as a computer program product **122**, for intelligent call graph generation is provided. In an alternative embodiment, program code **118** containing the computer-implemented process may be stored within computer readable media **120** as computer program product **122**. In another illustrative embodiment, the process for intelligent call graph generation may be implemented in an apparatus comprising a communications fabric, a memory connected to the communications fabric, wherein the memory contains computer executable program code, a communications unit connected to the communications fabric, an input/output unit connected to the communications fabric, a display connected to the communications fabric, and a processor

unit connected to the communications fabric. The processor unit of the apparatus executes the computer executable program code to direct the apparatus to perform the process.

[0035] With reference to **Figure 2,** a block diagram of a compilation system in accordance with various embodiments of the disclosure is presented. Compilation system **200** is an example of a compilation system that is enhanced to perform intelligent generation of a call graph. Compilation system **200** may be implemented as an addition to or within a current compilation system or as a standalone system in communication with a compiler system and associated resources. A call graph created using the compilation system **200** may be referred to as a compact call graph.

[0036] Compilation system **200** comprises a number of components focused on static analysis tasks including but not limited to a static analysis engine **202**, input in the form of program source code **204**, and providing output in the form of call graph **206**. Static analysis engine **202** may be implemented within a portion of an existing compiler system or as separate but related component. Static analysis engine **202** contains a number of cooperating components including data flow engine **208**, configuration options **210**, entry point selector **212** and branch eliminator **214**.

[0037] Data flow engine **208** provides a static data flow analysis capability for analyzing program source code **204** in a static mode. Data flow analysis provides identifying of functions and variables within the program code being evaluated. The data flow analysis also provides information with respect to code dependencies and function call relationships. Data flow engine **208** identifies and provides additional information to augment the information of the source code. The information includes reaching definitions and conditional branching information and other source code related information describing a context of use for the functions and variables defined within the source code.

[0038] Configuration options **210** provide a capability to provide direction to the process of analysis and build. For example, configuration options **210** may be configured to provide a preliminary exclusion list based on previous heuristics for a source code. In one example, a one

way dependence may be noted for use of Java™ library code. The one way use implies the java code will not call back or invoke the calling code. Once the call to the Java code occurs there is no need to have further branches. In another example, Java Class Library will change very little during minor versions and information can be provided to static analysis engine **202** that will allow it to prune unnecessary Java Class Library branches.

[0039] The specification of elements within configuration options **210** provides a granularity enabling methods within libraries to be selected rather than the library as a whole. For example, when a library provides a set of business functions, a specific function or method may be identified. Identification of the specific function or method avoids the specific function or method while allowing the remaining functions or methods in the library to be included.

[0040] Entry point selector **212** provides a capability to identify a specific entry point within program source code **204** rather than always starting at a main method. An entry point may be specifically identified in configuration options **210** or provided at the start of the analysis process. The entry point is a point of interest from which the call graph may be built. Typically in previous implementations an entry point was a point for which there was no interest other than a place to start. For example, when a desire to examine a particular function exists the function of interest may be searched for as the entry point. Rather than starting at the beginning of program source code **204**, a more efficient process would use entry point selector **212** to start at the particular function within program source code **204** to produce a call graph.

[0041] Branch eliminator **214** provides a capability to exclude or eliminate branches, paths, functions, variables, types or other elements from analysis and build operations. Branch eliminator **214** uses information from data flow engine **208** to examine dependencies in context to identify branches that are not needed, or reaching definitions that are not possible and should be ignored.

[0042] Build information is gathered from the results of analysis and configuration options and provided to call graph generator **216**. Call graph generator **216** provides call graph **206** as output as an optimized call graph also referred to as a compact call graph. An optimized call graph or

compact call graph in this scenario means the call graph generated is typically more specific and narrow in scope than would have otherwise been generated. Increased specificity and reduced scope of the call graph means less time and resources are required to generate a call graph because the generated call graph contains fewer elements and is typically smaller in size than a call graph created using a traditional process.

[0043] With reference to **Figure 3**, a block diagram of a call graph output from the compilation system of **Figure 2** in accordance with one embodiment of the disclosure is presented. Call graph **306** is an example of a compact call graph generated using compilation system **200** of **Figure 2**.

[0044] By identifying what is being searched for during a static analysis operation, many unnecessary branches can be eliminated very easily from a target call graph. Many underlying technologies and libraries are static and do not change. Using this static information, unimportant branches can be removed from the call graph reducing the size especially when the majority of the call graph is populated by superfluous nodes. As an example, the Java Class Library will change very little during minor versions. The information can be provided to static analysis engine **202** to enable elimination of unnecessary Java Class Library branches in a generated call graph.

[0045] For example, when using a standard open source widget toolkit to perform graphical operations, the program must have control of a shell. The shell is an abstraction of operating system resources and these resources must be eventually released during the course of running an application. A common pattern of using the open source widget toolkit is to create a new shell, perform some graphical operation and then eventually close the shell. If the program does not release the resources that the shell contains, not only is there a memory leak on the Java side but there is also a resource leak on the operating system side where certain resources are never recycled for the lifespan of the application. Since it is known that the Java Class Library does not, under any circumstance, close and release a shell, it is therefore possible to prune, or better avoid creating, any Java Class Library branches within the call graph and improve analysis times dramatically.

[0046] In call graph **300** node **302** represents the creation of an open source widget toolkit shell and node **304** represents a method call to one of Java Class Library nodes (such as the creation of an Object or a Collection -- these nodes are irrelevant to the analysis). Without a pruning technique, there are 7 nodes to analyze. Using the pruning technique, node **304** is identified as unimportant for further analysis and is eliminated along with associated dependent nodes. Call graph **306** has only 3 nodes left to analyze, comprising node **302** and adjacent nodes on the same path as node **302**. The nodes of a call graph portion **308** containing nodes adjacent to node **304** have been eliminated.

[0047] Call graph **300** and call graph **306** have been shown for illustration purposes, only and would not have been created using the process of compilation system **200** of **Figure 2**. Using the process of the process of compilation system **200** of **Figure 2**, only call graph **306** would be provided as output.

[0048] With reference to **Figure 4**, a flowchart of a call graph generating process using the compilation system of **Figure 2**, in accordance with one embodiment of the disclosure is presented. Process **400** is an example of a call graph generation process using static analysis engine **202** of compilation system **200** of **Figure 2**.

[0049] Process **400** starts (step **402**) and receives a request to build a call graph to form a received request (step **404**). Process **400** also receives source data associated with the received request (step **406**). The received source code is typically in the form of a program, application or code portion to be analyzed. For example, the source code may represent a portion of application that is being examined to determine a source of memory leakage. The code portion need not be a complete application or program.

[0050] Process **400** receives configuration options associated with the received request (step **408**). The configuration options may be specific to the source code, the request or both. The configuration options may also be generic and used as a base from which to adjust further examination. In some cases a set of default configuration options may be used. The configuration options may be provided in a previously stored file or data set or as responses to a set of user interface prompts.

[0051] Process **400** determines whether an entry point for the call graph is identified (step **410**). Identifying an entry point typically includes searching for predetermined elements in the source code. In another embodiment, the configuration option may specify an entry point as a named element, for example a function name. When a determination is made that an entry point for the call graph is identified, a "yes" result is obtained. When a determination is made that an entry point for the call graph is not identified, a "no" result is obtained. When a "no" result is obtained in step **410**, process **400** skips ahead to perform step **422**. When a "yes" result is obtained in step **410**, process **400** analyzes dependencies (step **412**). Dependencies indicate conditional flow of operations in the stream of program source code.

[0052] From the dependency analysis, and configuration options, process **400** identifies exclusions (step **414**). Exclusions are nodes, and paths or branches that are not required to be in the generated call graph. Elimination of the elements identified now reduces the time and resource to generate the call graph. The analysis determines what elements should be in the call graph output by avoiding any unnecessary branches and nodes in the generation input. The analysis enables a target function to be included while avoiding function that is not desired. Identifying what to search for reduces the size of the generated output by reducing the input to the generation operation. Known behavior of the source code may also be used to identify elements to avoid, as in the case of the one way Java library call.

[0053] Process **400** determines whether analysis required to build a compact call graph is complete (step **416**). When a determination is made that analysis required to build a compact call graph is complete, a "yes" result is obtained. When a determination is made that analysis required to build a compact call graph is not complete, a "no" result is obtained. When a "no" result is obtained in step **416**, process **400** loops back to step **412** to continue to perform analysis required to build a compact call graph.

[0054] When a "yes" result is obtained in step **416**, process **400** generates a requested compact call graph (step **418**). The generated call graph is created specific to the selected entry point and within the constraints of the scope defined as a result of the dependency analysis and exclusion

identifications. Because the generated call graph contains only the elements of interest, while unnecessary elements have been avoided in the generation process, the generated call graph is referred to as a compact call graph. Compact in this context means the generated call graph contains less elements than would have been generated using a traditional process. A traditional process requires "pruning" after the generation was complete.

[0055] Process **400** returns the generated compact call graph to a requester (step **420**) with process **400** terminating thereafter (step **422**). Thus the process just described typically reduces the size of a generated call graph to form a compact call graph by avoiding the generation of extraneous nodes. Extraneous nodes are not pruned from the call graph after the call graph has been generated. Rather extraneous nodes and branches are not generated, thus avoiding the overhead of creating data elements that are not wanted. Process **400** thus provides an example an example of "pruning on the fly" to create a compact call graph in a first instance. Process **400** avoids the creation of entries that would only later be discarded using a typical pruning process of previous call graph generation implementations. Process **400** therefore generates a compact call graph after completion of call graph analysis for the generation portion of the process. In contrast a traditional generation process creates a call graph that requires pruning after generation.

[0056] In one embodiment, a computer-implemented process for intelligent call graph generation receives a request to build a call graph to form a received request, receives a source code associated with the received request, receives configuration options, and determines whether an entry point for the call graph is identified in the source code. Responsive to a determination that the entry point for the call graph is identified, analyzes dependencies in the source code, and identifies exclusions in the source code using the analyzed dependencies to form identified exclusions, and generates the call graph without the identified exclusions.

[0057] The flowchart and block diagrams in the figures illustrate the architecture, functionality and operation of possible implementations of systems, methods, and computer program products according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of code, which

comprises one or more executable instructions for implementing a specified logical function. It should also be noted that, in some alternative implementations, the functions noted in the block might occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

[0058] The corresponding structures, materials, acts, and equivalents of all means or step plus function elements in the claims below are intended to include any structure, material, or act for performing the function in combination with other claimed elements as specifically claimed. The description of the present invention has been presented for purposes of illustration and description, but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope and spirit of the invention. The embodiment was chosen and described in order to best explain the principles of the invention and the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

[0059] The invention can take the form of an entirely hardware embodiment, an entirely software embodiment or an embodiment containing both hardware and software elements. In a preferred embodiment, the invention is implemented in software, which includes but is not limited to firmware, resident software, microcode, and other software media that may be recognized by one skilled in the art.

[0060] It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the processes of the present invention are capable of being distributed in the form of a computer readable medium of instructions and a variety of forms and that the present invention

applies equally regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media include recordable-type media, such as a floppy disk, a hard disk drive, a RAM, CD-ROMs, DVD-ROMs, and transmission-type media, such as digital and analog communications links, wired or wireless communications links using transmission forms, such as, for example, radio frequency and light wave transmissions. The computer readable media may take the form of coded formats that are decoded for actual use in a particular data processing system.

[0061] A data processing system suitable for storing and/or executing program code will include at least one processor coupled directly or indirectly to memory elements through a system bus. The memory elements can include local memory employed during actual execution of the program code, bulk storage, and cache memories which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

[0062] Input/output or I/O devices (including but not limited to keyboards, displays, pointing devices, etc.) can be coupled to the system either directly or through intervening I/O controllers.

[0063] Network adapters may also be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the currently available types of network adapters.

[0064] The description of the present invention has been presented for purposes of illustration and description, and is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. The embodiment was chosen and described in order to best explain the principles of the invention, the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

CLAIMS:

What is claimed is:

1. A computer-implemented process for intelligent call graph generation, the computer-implemented process comprising:

receiving a request to build a call graph to form a received request;

receiving a source code associated with the received request;

receiving configuration options;

determining whether an entry point for the call graph is identified in the source code;

responsive to a determination that the entry point for the call graph is identified, analyzing dependencies in the source code;

identifying exclusions in the source code using the analyzed dependencies to form identified exclusions;

determining whether analysis required to build a compact call graph is complete; and

responsive to a determination that analysis required to build a compact call graph is complete, generating the compact call graph without the identified exclusions.

2. The computer-implemented process of claim 1 wherein receiving configuration options further comprises:

receiving configuration options from a file, from a user response to a prompt in a user interface or a combination thereof.

3. The computer-implemented process of claim 1 wherein the configuration options comprise a set of options including a preliminary exclusion list based on previous heuristics for the source code and selection criteria for selecting functions, function types, libraries and members within libraries.

4. The computer-implemented process of claim 1 wherein determining whether an entry point for the call graph is identified in the source code further comprises:

receiving a configuration option specifying an entry point as a named element; and

searching for the named element in the source code.

5. The computer-implemented process of claim 1 wherein analyzing dependencies in the source code further comprises:

receiving additional information provided from a data flow engine to augment the information of the source code, wherein the information includes reaching definitions and conditional branching information providing a context of use for the source code.

6. The computer-implemented process of claim 3 wherein identifying exclusions in the source code using the analyzed dependencies to form identified exclusions further comprises:

combining the preliminary exclusion list form the configuration options with the analyzed dependencies.

7. The computer-implemented process of claim 1 wherein generating the call graph without the identified exclusions further comprises:

returning the generated compact call graph to a requester.

8. A computer program product for intelligent call graph generation, the computer program product comprising:

a computer recordable-type media containing computer executable program code stored thereon, the computer executable program code comprising:

computer executable program code for receiving a request to build a call graph to form a received request;

computer executable program code for receiving a source code associated with the received request;

computer executable program code for receiving configuration options;

computer executable program code for determining whether an entry point for the call graph is identified in the source code;

computer executable program code responsive to a determination that the entry point for the call graph is identified, for analyzing dependencies in the source code;

computer executable program code for identifying exclusions in the source code using the analyzed dependencies to form identified exclusions;

computer executable program code for determining whether analysis required to build a compact call graph is complete; and

computer executable program code responsive to a determination that analysis required to build a compact call graph is complete, for generating the compact call graph without the identified exclusions.

9. The computer program product of claim 8 wherein computer executable program code for receiving configuration options further comprises:

computer executable program code for receiving configuration options from a file, from a user response to a prompt in a user interface or a combination thereof.

10. The computer program product of claim 8 wherein the configuration options comprise a set of options including a preliminary exclusion list based on previous heuristics for the source code and selection criteria for selecting functions, function types, libraries and members within libraries.

11. The computer program product of claim 8 wherein computer executable program code for determining whether an entry point for the call graph is identified in the source code further comprises:

computer executable program code for receiving a configuration option specifying an entry point as a named element; and

computer executable program code for searching for the named element in the source code.

12. The computer program product of claim 8 wherein computer executable program code for analyzing dependencies in the source code further comprises:

computer executable program code for receiving additional information provided from a data flow engine to augment the information of the source code, wherein the information

includes reaching definitions and conditional branching information providing a context of use for the source code.

13. The computer program product of claim 10 wherein computer executable program code for identifying exclusions in the source code using the analyzed dependencies to form identified exclusions further comprises:

computer executable program code for combining the preliminary exclusion list form the configuration options with the analyzed dependencies.

14. The computer program product of claim 8 wherein computer executable program code for generating the call graph without the identified exclusions further comprises:

computer executable program code for returning the generated compact call graph to a requester.

15. An apparatus for intelligent call graph generation, the apparatus comprising:

a communications fabric;

a memory connected to the communications fabric, wherein the memory contains computer executable program code;

a communications unit connected to the communications fabric;

an input/output unit connected to the communications fabric;

a display connected to the communications fabric; and

a processor unit connected to the communications fabric, wherein the processor unit executes the computer executable program code to direct the apparatus to:

receive a request to build a call graph to form a received request;

receive a source code associated with the received request;

receive configuration options;

determine whether an entry point for the call graph is identified in the source code;

responsive to a determination that the entry point for the call graph is identified, analyze dependencies in the source code;

identify exclusions in the source code using the analyzed dependencies to form identified exclusions;

determine whether analysis required to build a compact call graph is complete; and

responsive to a determination that analysis required to build a compact call graph is complete, generate the compact call graph without the identified exclusions.

16. The apparatus of claim 15 wherein the processor unit executes the computer executable program code to receive configuration options further directs the apparatus to:

receive configuration options from a file, from a user response to a prompt in a user interface or a combination thereof.

17. The apparatus of claim 15 wherein the configuration options comprise a set of options including a preliminary exclusion list based on previous heuristics for the source code and selection criteria for selecting functions, function types, libraries and members within libraries.

18. The apparatus of claim 15 wherein the processor unit executes the computer executable program code to determine whether an entry point for the call graph is identified in the source code further directs the apparatus to:

receive a configuration option specifying an entry point as a named element; and

search for the named element in the source code.

19. The apparatus of claim 15 wherein the processor unit executes the computer executable program code to analyze dependencies in the source code further directs the apparatus to:

receive additional information provided from a data flow engine to augment the information of the source code, wherein the information includes reaching definitions and conditional branching information providing a context of use for the source code.

20. The apparatus of claim 17 wherein the processor unit executes the computer executable program code to identify exclusions in the source code using the analyzed dependencies to form identified exclusions further directs the apparatus to:

combine the preliminary exclusion list from the configuration options with the analyzed dependencies.

FIG. 1

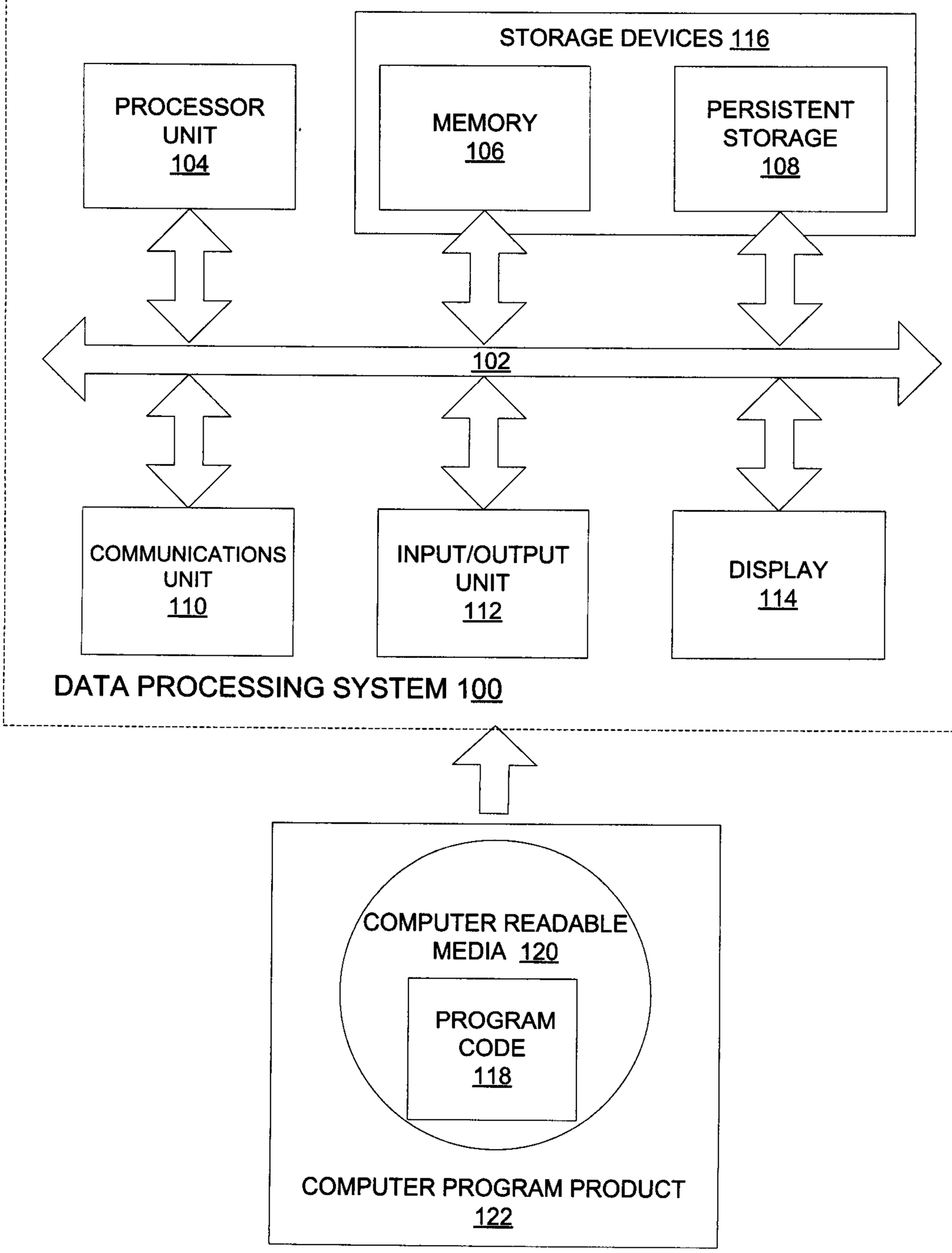

STORAGE DEVICES 116
PROCESSOR UNIT 104
MEMORY 106
PERSISTENT STORAGE 108
102
COMMUNICATIONS UNIT 110
INPUT/OUTPUT UNIT 112
DISPLAY 114
DATA PROCESSING SYSTEM 100
COMPUTER READABLE MEDIA 120
PROGRAM CODE 118
COMPUTER PROGRAM PRODUCT 122

FIG. 2

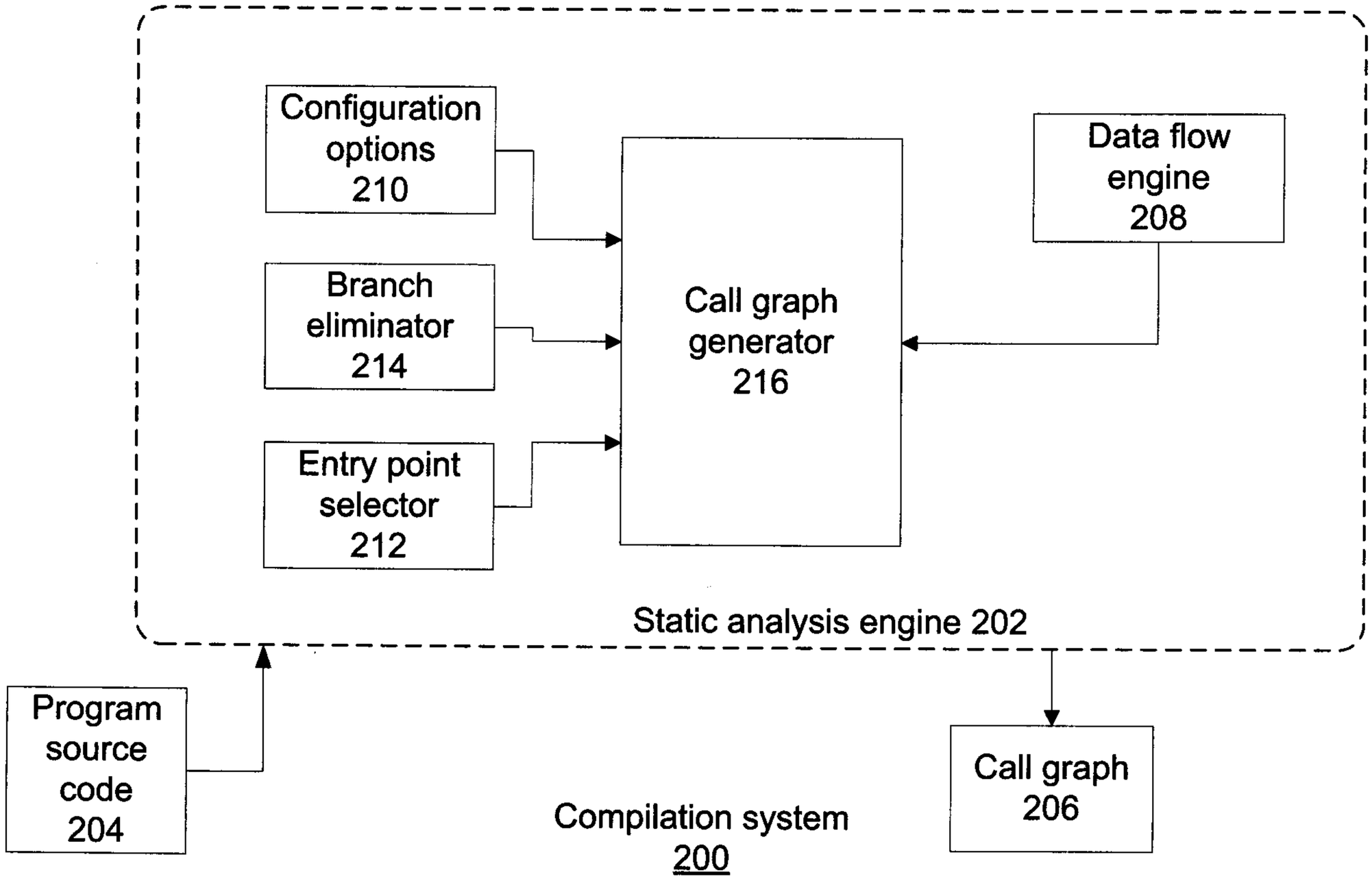

Configuration options 210
Branch eliminator 214
Entry point selector 212
Call graph generator 216
Data flow engine 208
Static analysis engine 202
Program source code 204
Call graph 206
Compilation system 200

FIG. 3

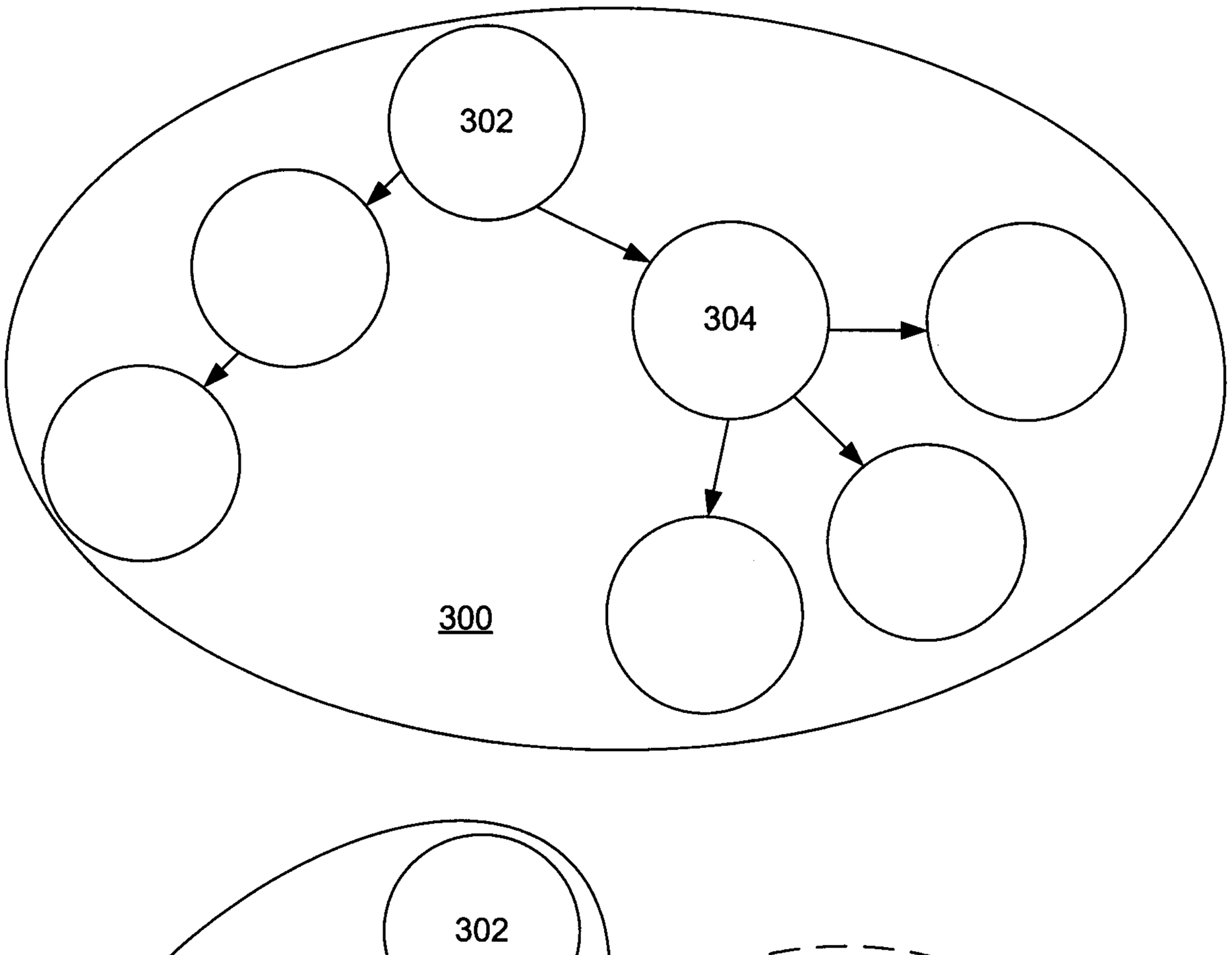
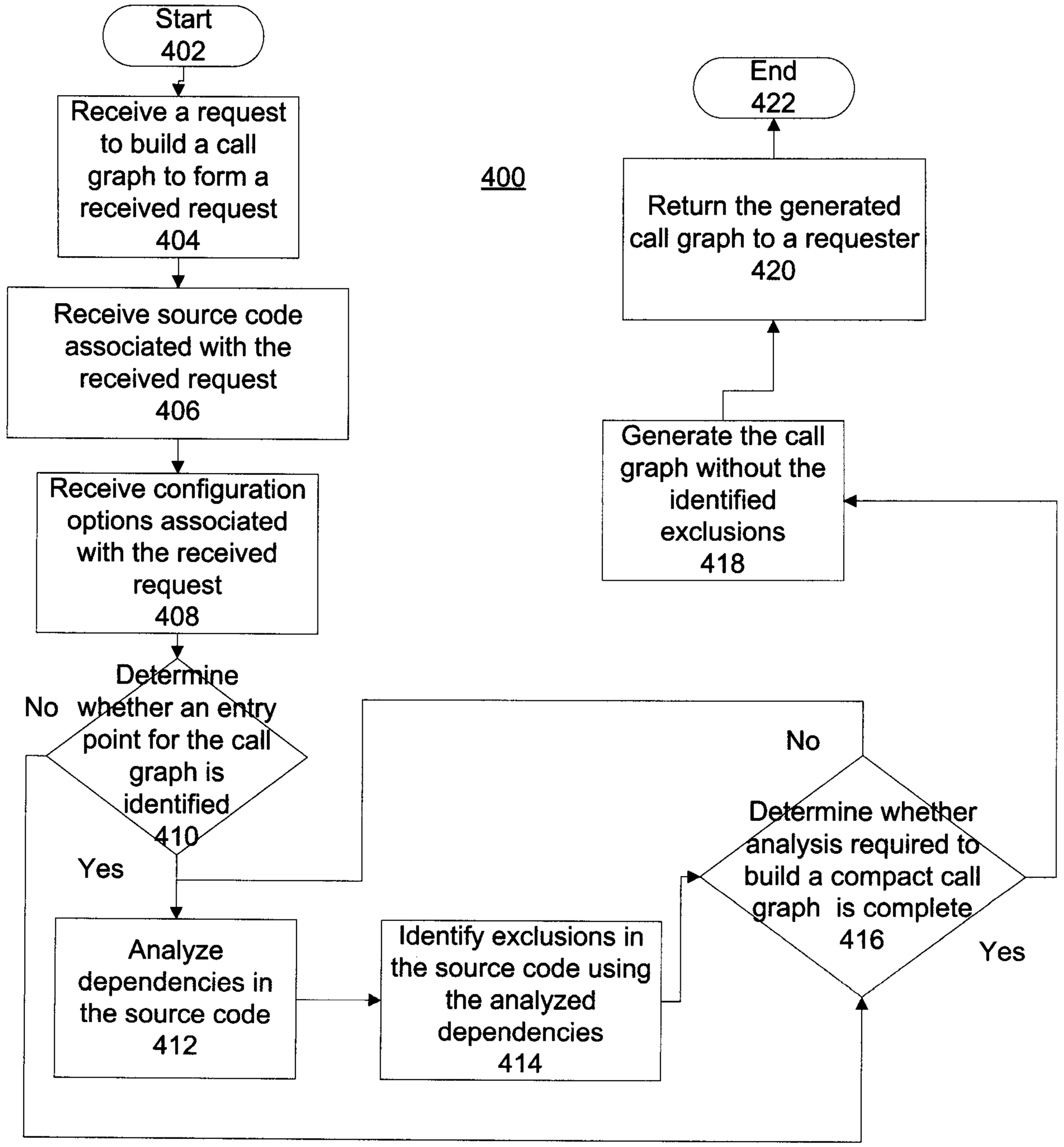


FIG. 4


Start
402
Receive a request to build a call graph to form a received request
404
Receive source code associated with the received request
406
Receive configuration options associated with the received request
408
Determine whether an entry point for the call graph is identified
410
No
Yes
Analyze dependencies in the source code
412
Identify exclusions in the source code using the analyzed dependencies
414
Determine whether analysis required to build a compact call graph is complete
416
No
Yes
Generate the call graph without the identified exclusions
418
Return the generated call graph to a requester
420
End
422
400

STORAGE DEVICES 116

PROCESSOR UNIT 104

MEMORY 106

PERSISTENT STORAGE 108

102

COMMUNICATIONS UNIT 110

INPUT/OUTPUT UNIT 112

DISPLAY 114

DATA PROCESSING SYSTEM 100

COMPUTER READABLE MEDIA 120

PROGRAM CODE 118

COMPUTER PROGRAM PRODUCT 122