(12) UK Patent Application (19) GB (11) 2503590 (13) A

(43) Date of A Publication 01.01.2014

(21) Application No: 1314586.7

(22) Date of Filing: 14.08.2013

(71) Applicant(s):
Micro Focus IP Development Ltd
The Lawn, 22-30 Old Bath Road, NEWBURY, Berkshire, RG14 1QN, United Kingdom

(72) Inventor(s):
Stephen Gennard

(74) Agent and/or Address for Service:
EIP
Fairfax House, 15 Fulwood Place, LONDON, WC1V 6HU, United Kingdom

(51) INT CL:
G06F 9/445 (2006.01) ***G06F 9/45*** (2006.01)

(56) Documents Cited:
US 20060225033 A1 **US 20050172299 A1**
US 20030182460 A1

(58) Field of Search:
INT CL **G06F**
Other: **ONLINE: EPODOC & WPI; TXTE; XPI3E, XPIPCOM and XPLNCS; Internet.**

(54) Title of the Invention: **Interoperability unit**
Abstract Title: **Interoperability bridge unit for dynamic linking of executable library functions to an application using platform invoke**

(57) An application program comprising executable code is deployed in an RTE (Runtime Environment) comprising an interoperability bridge unit 405. A reference to a function, originally instructed in a system-independent programming language, is received from the executable code 450 and the reference resolved to determine the function name. The executable code is derived from an intermediate language code, e.g. bytecode. If the function name is located in an export portion of an executable library 465 registered with the RTE, typically a DLL (Dynamic Link Library) then program code, comprising a platform invoke statement referencing the library and function name, is dynamically generated, prepared and executed to invoke the function. A function code detector 415 receives the reference and a function locator resolves the reference to determine the function name. A code generator 435 dynamically generates program code 445 comprising the platform invoke statement for the function and this code is passed to a code implementer 455 which prepares the executable code. A static arrangement is also disclosed in which source code is parsed by a compiler. Program code containing a platform invoke statement for a function can then be generated and compiled to intermediate language code before run time.

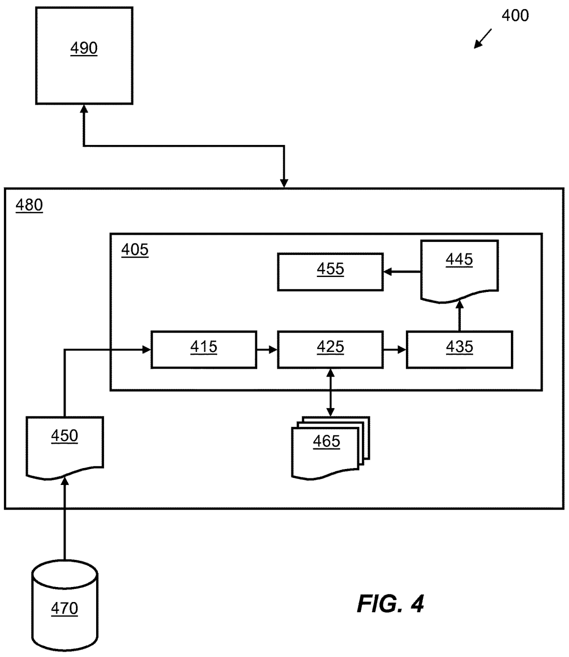


FIG. 4

GB 2503590 A

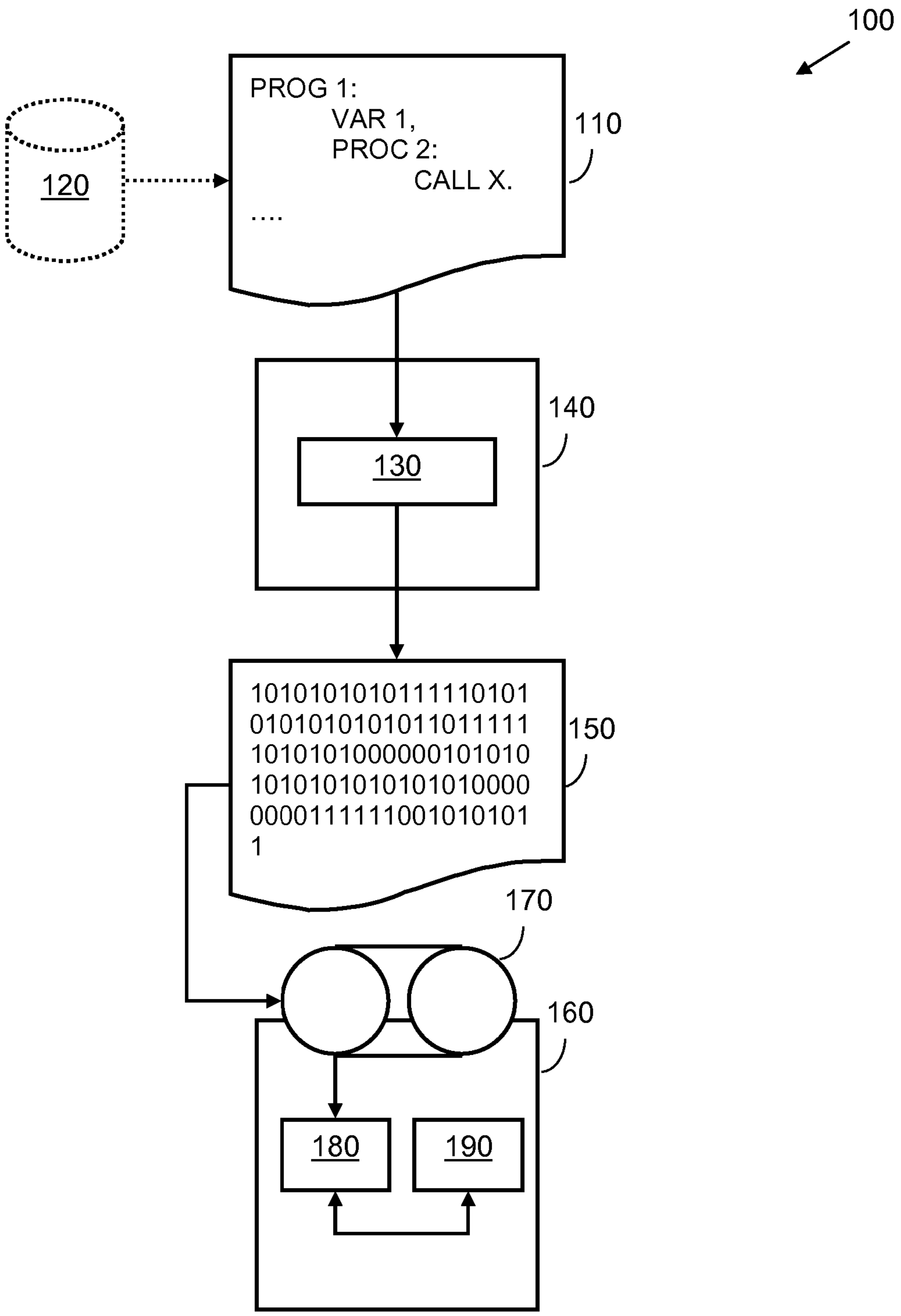
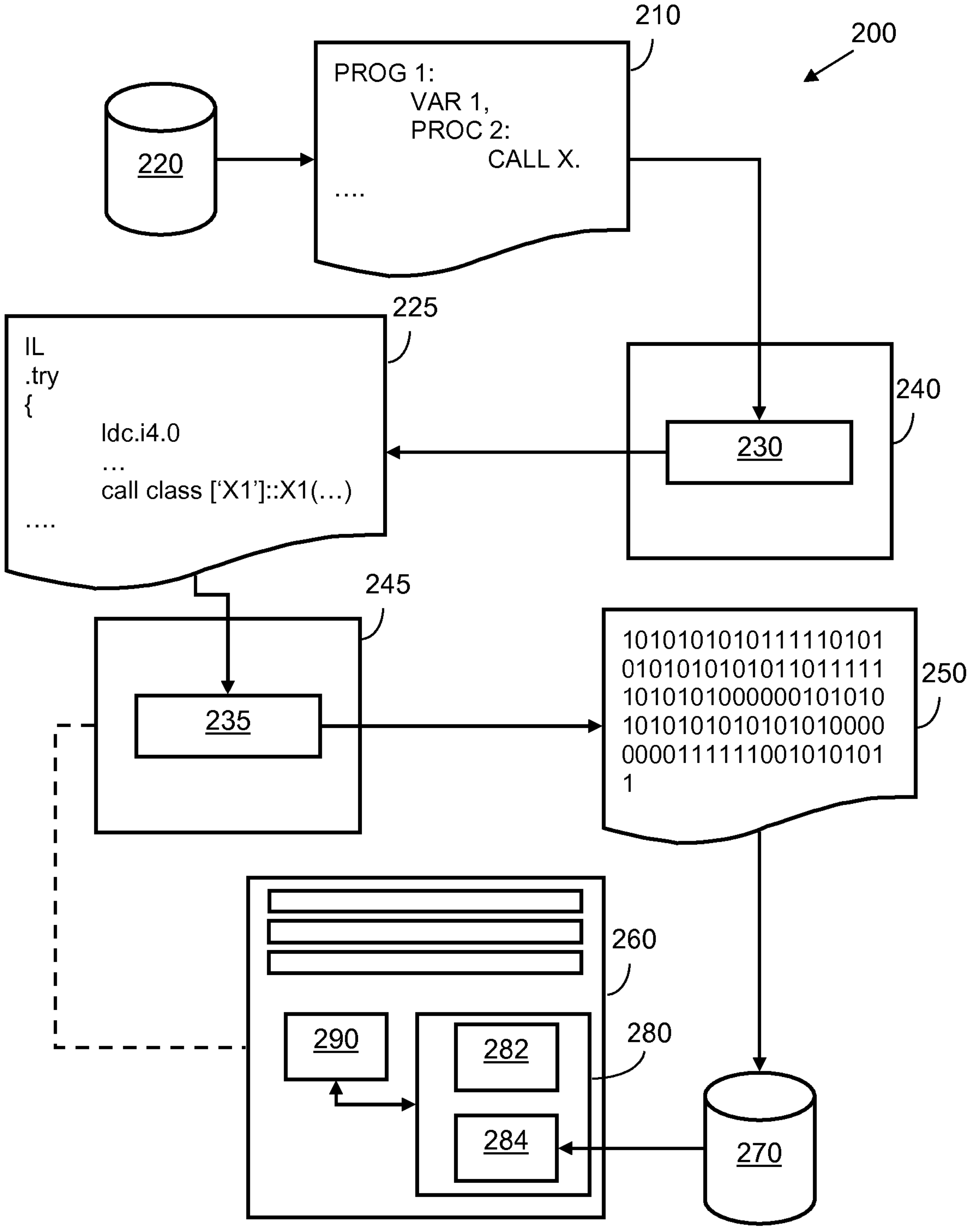
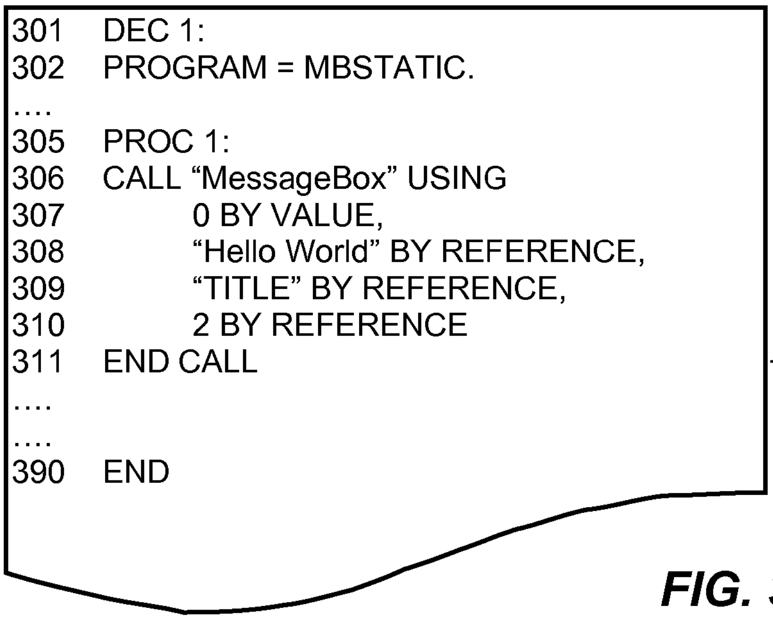

100
PROG 1:
VAR 1,
PROC 2:
CALL X.
....
110
120
140
130
150
10101010101011110101
01010101010110111111
10101010000000101010
10101010101010100000
00000111111001010101
1
170
160
180
190

FIG. 1



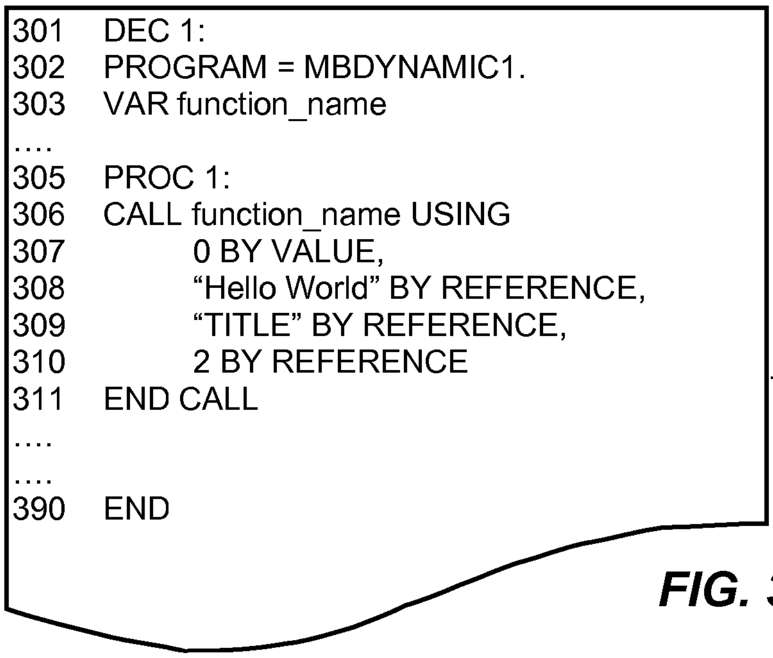
200
210
PROG 1:
VAR 1,
PROC 2:
CALL X.
….
220
225
IL
.try
{
ldc.i4.0
…
call class [‘X1’]::X1(…)
….
240
230
245
235
250
10101010101011110101
01010101010110111111
10101010000000101010
10101010101010100000
00001111110010101011
1
260
280
282
284
290
270

FIG. 2



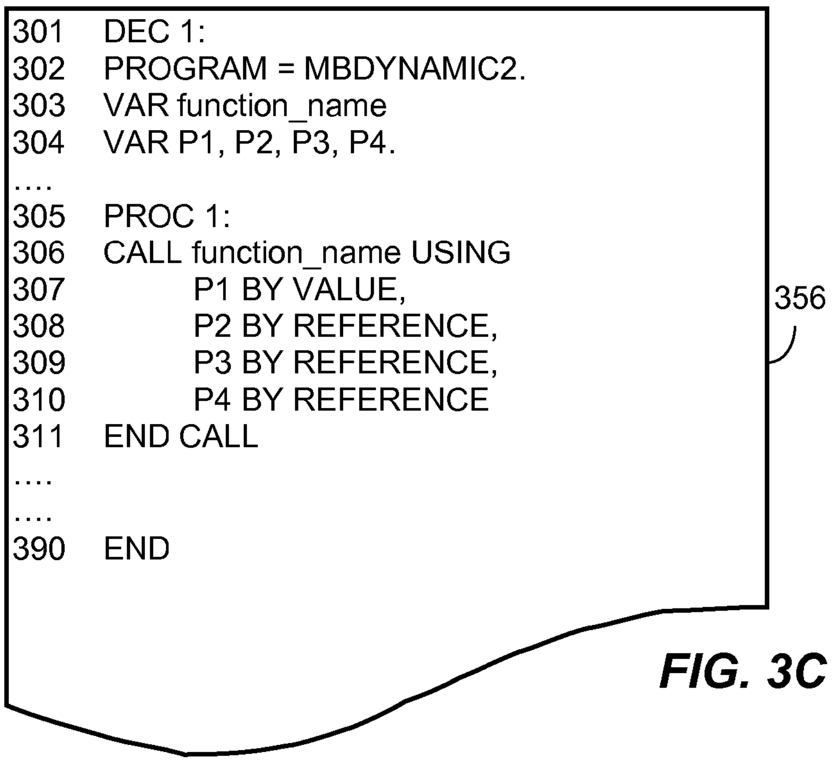
```
301   DEC 1:
302   PROGRAM = MBSTATIC.
....
305   PROC 1:
306   CALL “MessageBox” USING
307           0 BY VALUE,
308           “Hello World” BY REFERENCE,
309           “TITLE” BY REFERENCE,
310           2 BY REFERENCE
311   END CALL
....
....
390   END
```

FIG. 3A



```
301   DEC 1:
302   PROGRAM = MBDYNAMIC1.
303   VAR function_name
....
305   PROC 1:
306   CALL function_name USING
307           0 BY VALUE,
308           “Hello World” BY REFERENCE,
309           “TITLE” BY REFERENCE,
310           2 BY REFERENCE
311   END CALL
....
....
390   END
```

FIG. 3B



```
DEC 1:
PROGRAM = MBDYNAMIC2.
VAR function_name
VAR P1, P2, P3, P4.
....
PROC 1:
CALL function_name USING
        P1 BY VALUE,
        P2 BY REFERENCE,
        P3 BY REFERENCE,
        P4 BY REFERENCE
END CALL
....
....
END
```

FIG. 3C

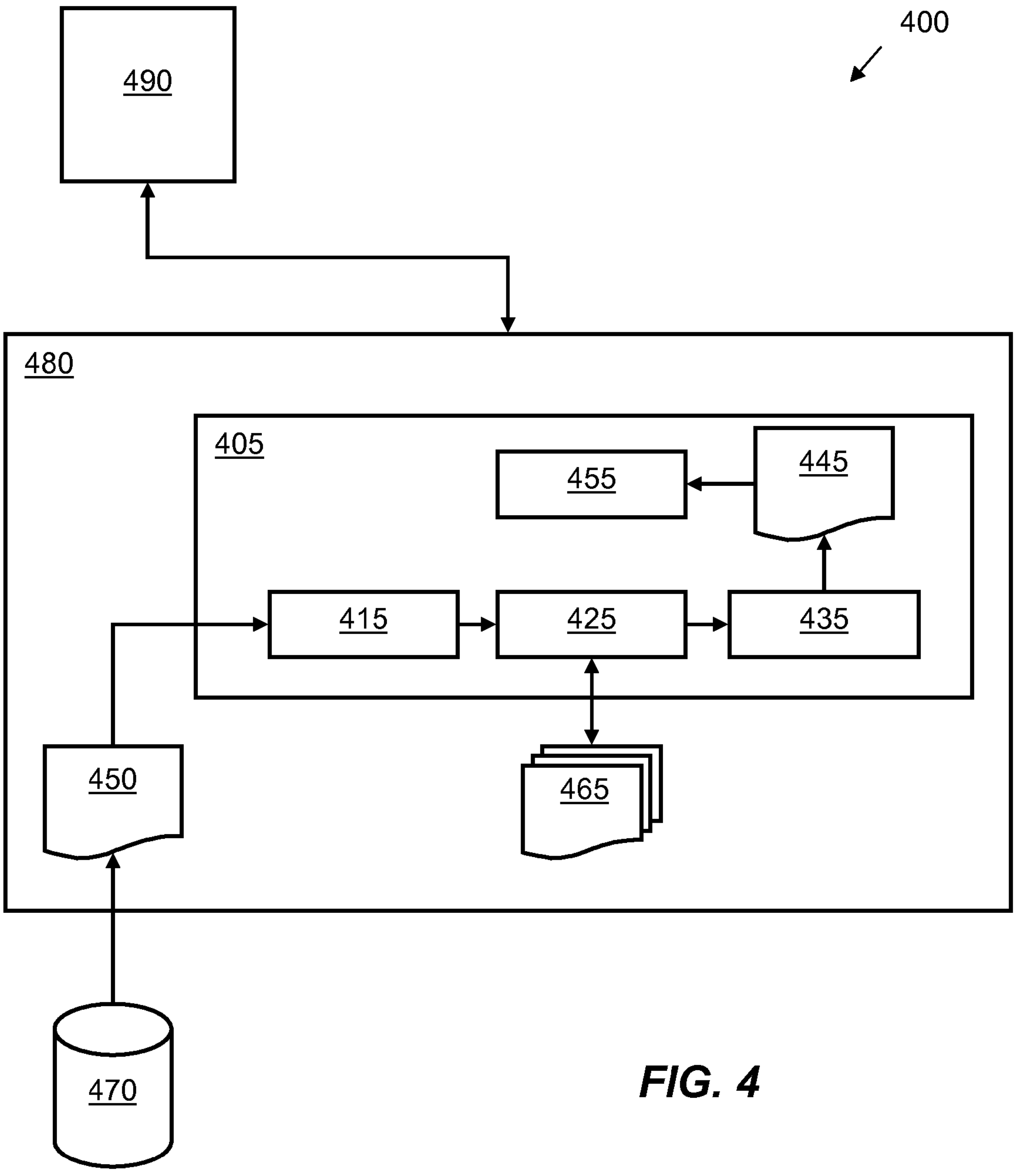


FIG. 4

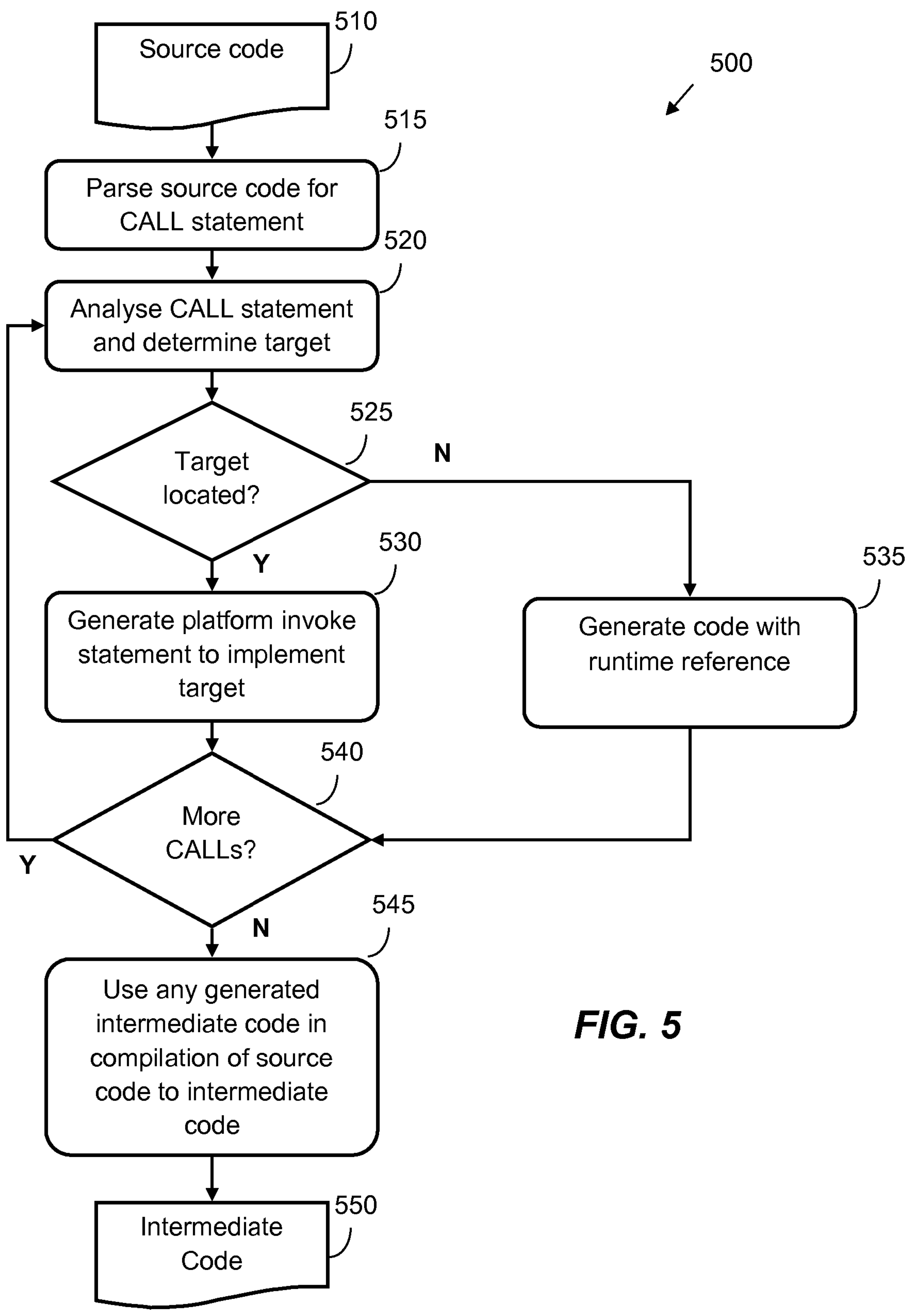
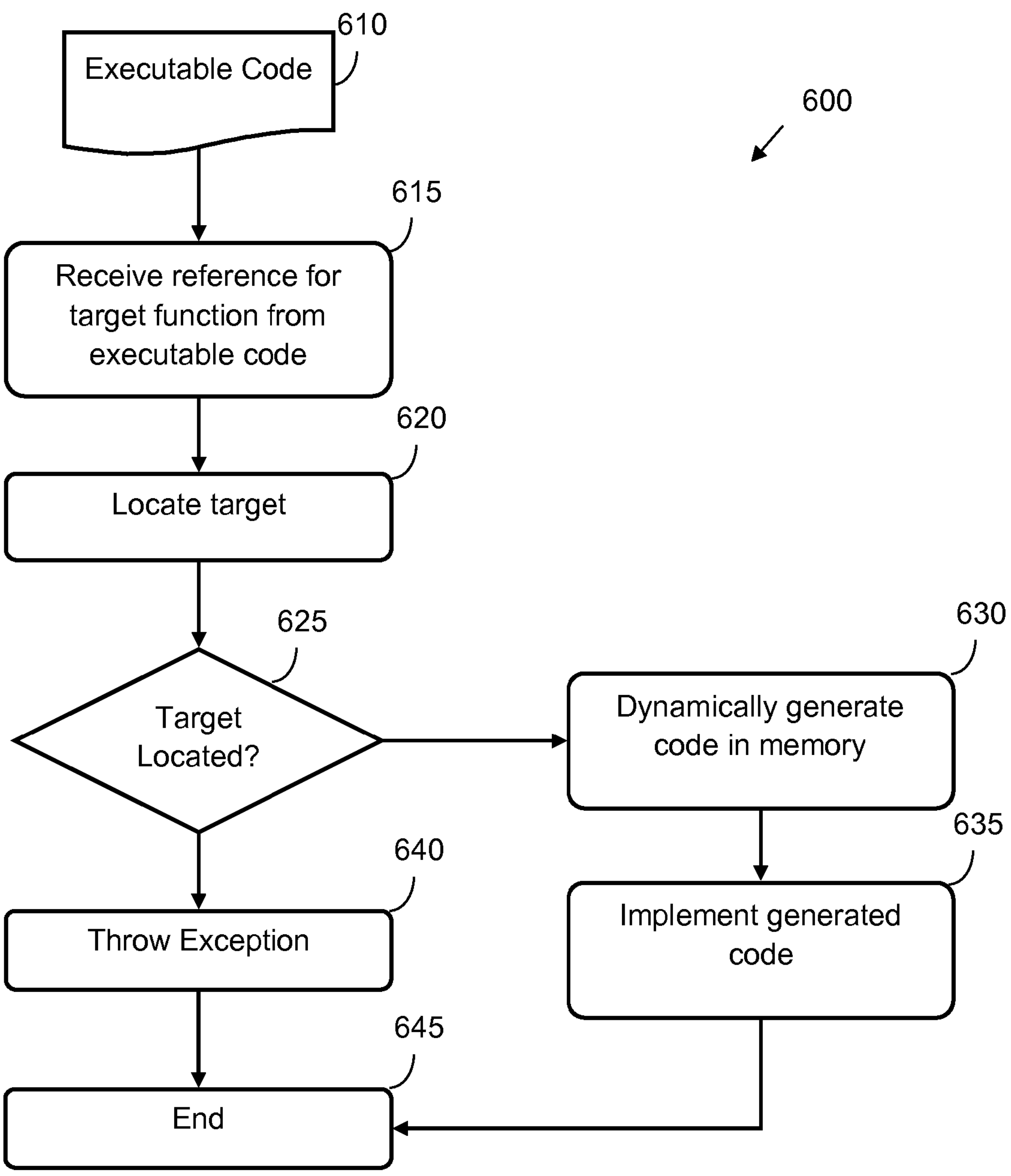

500
510
Source code
515
Parse source code for CALL statement
520
Analyse CALL statement and determine target
525
Target located?
N
Y
530
Generate platform invoke statement to implement target
535
Generate code with runtime reference
540
More CALLs?
Y
N
545
Use any generated intermediate code in compilation of source code to intermediate code
550
Intermediate Code

FIG. 5



610
Executable Code
600
615
Receive reference for target function from executable code
620
Locate target
625
Target Located?
630
Dynamically generate code in memory
635
Implement generated code
640
Throw Exception
645
End

FIG. 6

INTEROPERABILITY UNIT

Technical Field

The present invention relates to one or more of preparation and deployment of computer program code upon a computing system. In particular, the present invention relates to improvements in how a given set of computer program code is translated into control instructions for a given computing system.

Background

There is often an inherent tension in computing technology between the pace of change of computer architectures and the need for stable, reliable solutions that operate for decades. Before personal computers entered the market in the 1980s and 1990s, the concept of "computing" as a technical field was associated with large-scale, centralised "mainframe" computing systems. These systems were often cabinet or room-sized and were used for mission-critical tasks in large organisations, scientific institutions and government. In the 1960s and 1970s many mainframe computing systems were manufactured and supplied by International Business Machines Corporation (IBM®). For example, well-known systems included the IBM 700/7000 series, and those based on the System/360 and System/370 computing architectures. Many features that are now taken for granted, such as transistor then complementary metal-oxide-semiconductor (CMOS) central processing units (CPUs), 8-bit bytes, 32-bit words, hardware floating point support and 32-bit words were introduced slowly over decades in these systems.

In parallel with the development of modern computing architectures based on mainframe innovations, there was a need to better control computing operations. Although difficult to comprehend in the modern era, the System/360 computing architecture was one of the first computers in wide use to include dedicated hardware provisions for the use of an operating system. Before that time, mainframe systems required computer program code to be manually initiated, often on punched cards. Early mainframe systems were controlled and programmed using system-specific machine code. Over time, mnemonic assembly languages developed, which added support for macro instructions. This then led into the development of the first system-independent programming languages, languages such as FORTRAN, LISP, COBOL and ALGOL 60 (so-called “third generation languages”). Grace Hopper, an early computing pioneer, developed the first compiler and later the COBOL language. Much of the work in this era paved for way for technical aspects of digital computing that are considered fundamental today, such as subroutines, formula translation, relative addressing, link loading, code optimization and symbolic manipulation. System-independent programming languages, and the compiler technology that allowed system-independent program code to be deployed as machine code to specific hardware architectures, greatly improved the portability of computer programs, benefiting the control of computing devices regardless of specific program content or function. It also enabled a computer program to continue to control a computer system as a computer architecture evolved. Hardware features could be added to the computer system or architecture without requiring a rewrite of computer program code.

The success of system-independent programming languages brings its own set of technical problems. As computer programs could survive upheaval in physical architectures, resources could be invested in building stable, mission-critical computing functions. These functions could be designed to be operational for decades, human lifetimes even. For example, over the years many control, data recordal and file handling functions were developed, embodied in computer program code and became the foundation of modern engineering and production systems. As more functionality was added, this was often achieved by appending additional computer program code to existing computer programs. At the turn of the modern Internet era in the twenty-first century, many organisations were reliant on computer program code originally written and developed in the 1960s and 1970s.

As hardware developments have accumulated and incorporated global networking aspects, modern computer systems are required to operate in an environment quite different from the early eras of mainframe computing. However, as described above, organisations, entities and control systems are deeply reliant on legacy computer program code in programming languages that date from these earlier eras. Migration is one option but may not be entirely possible. Legacy computer programs can run to millions of lines of computer program code, contributed by thousands of different programmers over tens of years. Early system-independent computing languages were often sequential in nature, making it difficult to identify and extract individual functions and often requiring a so-called "big bang" migration where all computing functions are transferred to a new implementation at once. Even though they are often rigorously tested, these tests cannot cover all of the complexity of the legacy computer program. For implementations in hospitals, civil infrastructure, and aviation, as just a few examples, the risk of failure may be too great to attempt migration. This then requires aligning the technical benefits of modern hardware systems with computer program code that may pre-date the implementing computer engineers.

Summary

Aspects of the present invention are set out in the appended claims.

Further features and advantages of the invention will become apparent from the following description of preferred embodiments of the invention, given by way of example only, which is made with reference to the accompanying drawings.

Brief Description of the Drawings

Figure 1 is a schematic diagram showing a first set of system components for controlling a mainframe computing device according to an example;

Figure 2 is a schematic diagram showing a second set of system components for controlling a computing device according to an example;

Figures 3A to 3C are schematic diagrams showing examples of computer program code in a system-independent programming language;

Figure 4 is a schematic diagram showing components of an interoperability bridge according to an example;

Figure 5 is a flow diagram showing a first method of linking managed and native computer program code according to an example; and

Figure 6 is a flow diagram showing a second method of linking managed and native computer program code according to an example.

Detailed Description

Certain examples described herein provide technical improvements when deploying legacy computer programs. For example, computer program code that was originally written in a computer programming language for use with legacy mainframe computer systems, such as those based on the IBM System/360 computing architecture, may be optimised for use with modern computing architectures. In this sense the optimisation is independent of clever programming techniques in the computer program itself; rather it is provided by adapting the control instructions that are used to operate the underlying computer hardware, i.e. a more efficient set of control instructions are provided that are independent of high-level program function. Optimisation may involve improving the speed at which a computer implements a program for a fixed CPU speed and/or making more efficient use of working memory, for example by reducing the amount of memory used and/or optimising memory read/write operations. Hence, the examples described herein make a computing system implementing any machine code that results from said examples work better, faster or more reliably in terms of its performance, the computing system is better controlled to provide measurable, deep-level improvements.

It is also possible to make use of technical functionality that was never available on legacy mainframe systems, for example modern Internet and networking functions that may be employed and linked in a suitable runtime environment. These advantages may be realised without need to alter the original legacy computer program code. This is of benefit in mission critical computing systems, for example in health-care or transportation, where it is not possible to modify the computer program code for fear of failure or error. As the original legacy program is not modified, a technical contribution provided by certain described examples is not provided within the computer program as such; indeed in mission critical computing systems functionality of the computer program code must not change. The examples do not relate to a better way to write a program; rather they provide a measurable improvement as compared with fundamental programming conventions.

Certain examples described herein are arranged to be implemented on one or more computing systems, for example a system comprising one or more processors and a memory, such as a random access memory. The examples described herein do not comprise a method that is implemented manually or in the mind; indeed this is impossible.

Native Environment

Figure 1 shows a first set of system components 100 for controlling a mainframe computer system 160 according to an example. Source code 110 comprises computer program code for a computer program written in a system-independent programming language, e.g. data in the form of command statements and/or variable declarations. The language may comprise one of FORTRAN, LISP, COBOL, PL/I and ALGOL 60. Source code 110 may be retrieved from a computer-readable storage medium 120. Source code 110 is typically human-readable. To control the mainframe computer system 160, the source code 110 must be converted to a machine-readable version. This is performed by a compiler unit 130 of a computer device 140. The compiler unit 130 takes source code 110 and converts it into object code 150. The object code comprises a version of the source code 110 that may be read and executed by mainframe computer system 160. The command statements and/or variable declarations in the source code 110 are converted into control instructions specific to the mainframe computer system 160, i.e. instructions that may be interpreted by the specific hardware components of the mainframe computer system. The object code 150 is typically not human-readable, e.g. Figure 1 shows a binary version of the control instructions. It is sometimes referred to as machine or executable code. In this case, the object code 150 is stored on a computer-readable storage medium 170. When the computer program is to be implemented the object code 150 is loaded from the computer-readable storage medium 170 into a memory 180 of the mainframe computer system 160. The memory 180 is typically a random access memory. During execution, control instructions as defined in the object code 150 are processed by one or more processors 190 of the mainframe computer system 160. Each processor set typically has a different set of suitable control instructions. An instruction may control operations such as, among others, load, jump, and arithmetic logic unit commands.

In mainframe computer systems from the 1960s, 70s and 80s, the memory and the one or more processors may be limited when compared to modern computer systems. For example, a state-of-the-art system used by the National Aeronautics and Space Administration (NASA) and installed in 1969 had a central processing unit cycle time of 60 nanoseconds (e.g. a speed of 16MHz) and a memory cycle time of 780 nanoseconds (e.g. around 1.3MHz). The memory size was 2MB. Source code 110 may be representative of programs written to run on these mainframe systems.

Managed Environment

The arrangement of Figure 1 may be referred to as a native environment, wherein source code is compiled to native code, i.e. object code 150 for the mainframe computer system 160. In contrast, Figure 2 shows an arrangement that may be referred to as a managed environment. In this case, source code 210 is compiled by a first compiler unit 230 of a first computer device 240. The first compiler unit 230 converts the source code 210 to intermediate code 225. Intermediate code is system-independent (e.g. processor and computer architecture independent). It is of a lower level than source code 210, i.e. its instructions have a closer mapping to system-specific instruction sets. Intermediate code 225 may comprise common intermediate language (CIL) code, as defined by the common language infrastructure (CLI) standard and implemented as part of a Microsoft® .NET or Mono framework, or Java® bytecode. Intermediate code 225 in the form of common intermediate language code may be assembled into a form of bytecode called CLI assembly. This may be performed by the first compiler unit 230. In Figure 2 the intermediate code 225 is further compiled by a second compiler unit 235 of a second computer device 245 into object code 250. The first and second computer devices and/or the first and second compiler units may, in some cases, form part of the same device or unit. The second compiler unit 235 may comprise a just-in-time and/or ahead-of-time compiler unit, e.g. a Java virtual machine or a common language runtime (CLR). In the former case, the intermediate code 225 is converted into object code 250 at runtime, this may occur in a piece-meal or line-by-line (i.e. interpreted) fashion; in the latter case, the intermediate code 225 is converted into object code 250 before execution. In this latter case, the object code 250 may be stored in a computer-readable medium 270 before execution. In both cases, the object code is similar to object code 150 and at least a portion of object code 284 is loaded into a memory 280 of computer system 260 for execution by at least one processor 290.

In Figure 2, computer system 260 may comprise a modern server computing device (i.e. modern relative to the mainframe computer system 160). By compiling to intermediate code 225, source code 210 may be optimised to make use of advanced features of the computer system 260. For example, one or more of security, threading and memory management may be improved by making use of run-time management services 282. Run-time management services 282 are implemented by system libraries that comprise executable code – they may form part of an execution or runtime environment. This executable code is loaded into memory 280 along with at least a portion of the object code 284 during execution. In one case, run-time management services 282 may comprise the second compiler unit 235 in the form of a just-in-time compiler, wherein the second computer device 245 comprises the computer system 260 (as indicated by the dashed line). In this case, the second compiler unit 235 may comprise a virtual machine in the form of a common language runtime in the .NET framework or a Java virtual machine. The virtual machine may then handle exceptions, garbage collection (i.e. efficient use of space in memory 280), variable type safety (i.e. a discrepancy between data types) and thread management (i.e. control of machine code being processed by the at least one processor 290). These functions may not be available on mainframe system 160, or at least may be implemented in an inefficient manner due to the age of the hardware of the mainframe system 160. Even in a case where intermediate code 225 is compiled ahead-of-time to object code 250, the same functions of the run-time management services 282 may be implemented by suitable code conversion from intermediate code to object code (e.g. a given intermediate code statement may be compiled in a particular manner for a particular server computing device to make use of hardware functionality of said device using appropriate machine code functions).

Certain embodiments described herein may be implemented in a managed environment as described in relation to Figure 2. In one case, source code may be compiled to intermediate code for execution by a just-in-time compiler (e.g. a virtual machine) within a runtime environment. In another case, intermediate code may be compiled ahead of time. In both case, the output may be referred to as executable code, as the code may be executed directly or by way of the just-in-time compiler. Compilation may be performed on a first computing device for execution within a runtime environment installed on a second computing device, wherein compilation prepares control instructions based on the source code that may make use of functions provided by the runtime environment. As such compilation is described as a preparation step for subsequent deployment into a runtime environment that generates a concrete output – executable code to control a computing device.

Function Call Command Statement

Figures 3A to 3C show three different examples of at least a portion of computer program code. The computer program code may comprise command statements and/or variable declarations that conform to a programming language specification, i.e. source code written in a programming language. The programming language may be human-readable and comprise one of FORTRAN, LISP, COBOL, PL/I and ALGOL 60.

Figure 3A shows a first portion 352 of computer program code that comprises a “CALL” command statement 306. The “CALL” command statement runs a subroutine specified by a function name that follows the command statement. For example, in Figure 3A the function name is set by a static string variable with a value “MessageBox”. Hence, the “CALL” command statement 306 passes programmatic control to another portion of computer program code that implements a function “MessageBox”. Lines 307 to 310 of the first portion 352 of Figure 3A specify parameters that are passed to the function. These parameters follow a “USING” statement. The manner in which the parameter is passed to the function is also provided, e.g. either by value or by reference in the present example. Line 311 ends the “CALL” command statement 306.

In a comparative example, a subroutine that implements the function “MessageBox” is declared and written in another portion of computer program code. This other portion of computer program code and the first portion 352 are then compiled to native (i.e. object) code and then a command line linking tool is used to link the two portions.

Figure 3B shows a second portion 354 of computer program code that comprises a “CALL” command statement 306 that references a dynamic rather than static function name. For example, in Figure 3B there are no punctuation marks around the string following the “CALL” portion of the statement. This indicates that the string represents a variable and the contents of the variable may vary during execution on at least one processor (i.e. at runtime). For example, line 303 of the second portion 354 declares "function_name" as a string variable.

Figure 3C shows a third portion 356 of computer program code that comprises a "CALL" command statement 306 that references a dynamic function name and that passes dynamic parameters to the function, i.e. parameter values of parameters P1, P2, P3, and P4 are determined at runtime rather than at the time the third portion 356 was written. The parameters P1 to P4 are declared as variables in line 304 and referenced in lines 307 to 310.

When using a managed environment as shown in Figure 2, a command line linking tool is not available. Further the referenced function may be compiled to object code whereas the referencing code is compiled to intermediate code. It is thus not possible to link the object code files as in the comparative example. Moreover, in the examples of Figures 3B and 3C the function name is not known at the time of a first compilation; the value of a function name variable is only set at runtime. Hence, it is not possible to specify a file name for referenced code ahead of the execution of the referencing code. There is thus a problem of linking managed and native code that is compounded when function names are dynamic.

In certain examples, it may also be desirable to instruct the execution of computer program code that has been written in a programming language that is different from the source language of the referencing computer program code. For example, source code written in one of FORTRAN, LISP, COBOL, PL/I and ALGOL 60 in past decades may need to use and/or interface with one or more functions written in a more recently developed programming language such as C#, PHP or Java®. The latter functions may relate to, for example, network functions in a protocol that did not exist at the time the original source code was written.

In certain circumstances a system implemented by one or more portions of computer program code may need to be ported from a native environment to a managed environment, e.g. adapted from the environment of Figure 1 to the environment of Figure 2. As described previously, this may be to take advantage of modern computer architectures and infrastructures. Alternatively, an original mainframe computing system may need to be decommissioned due to aging and/or failing hardware. In this case it may not be possible to migrate portions of a system. For example, source code associated with these portions may have dependencies on third party libraries that are not supported within the managed environment. Alternatively, it may be too risky to port a large system in one go – this could lead to an unacceptable chance of error in mission critical applications such as hospitals or air traffic control systems.

Embodiments of the invention as discussed herein solve these problems by providing at least one of an adapted compiler system and an adapted runtime environment. These adapted systems enable an application program to be implemented by a combination of managed code and native code without modifying an original set of source code.

Interoperability Unit Example

Figure 4 shows a computer system 400 that comprises a memory 480 communicatively coupled to at least one processor 490. The memory 480 and the at least one processor 490 may comprise the memory 280 and processor 280 as shown in Figure 2. Figure 4 further shows an interoperability bridge unit 405 that forms part of a runtime environment. The runtime environment supports execution of a first set of executable code 450 by way of the at least one processor and the memory. The first set of executable code 450 comprises managed code, i.e. code derived from an intermediate language. For example, it may comprise assembled CIL, the assembled CIL being compiled from the third generation code shown in Figures 3A to 3C. The first set of executable code 450 in Figure 4 is loaded from a computer readable storage medium, in this case a hard disk drive 470. The interoperability bridge unit 405 may be implemented by way of computer program code (e.g. compiled C# code) that is loaded into memory 480 and executed by the at least one processor 490. The first set of executable code 450 may be representative of an application program originally written as source code 110, 210 or 352 to 356. The interoperability bridge unit 405 is arranged to, when a first set of executable code is implemented in the runtime environment, implement a second set of executable code for the computer system in the runtime environment to provide a function instructed in the system-independent programming language. The second set of executable code being may comprise code that is directly compiled from computer program code written in the system-independent programming language; for example, native or machine code derived from third generation code. It may alternative comprise a function provided by a standard library.

The interoperability bridge unit 405 comprises a function call detector 415, a function locator 425, a code generator 435 and a code implementer 455. The function call detector 415 receives a reference to the function in the first set of executable code. For example, as described in more detail below, during compilation of the first set of executable code every "CALL" command statement may be replaced in the intermediate language with a call to the function call detector 425. The call to the function call detector 425 passes a call-convention parameter, a variable representing the name of the function to "CALL" and any parameters that need to be passed to the function. The value of the variable representing the name of the function to "CALL" may be set at runtime in the cases of Figures 3B and 3C, i.e. a value is determined as the first set of executable code is being processed by processor 490. The function call detector 415 is communicatively coupled to the function locator 425. The function call detector 415 passes at least the variable representing the name of the function to the function locator 425. The function locator 425 resolves this variable to determine the name of the function. This may comprise extracting a relevant portion of a string and saving the string in memory. The function locator 425 is then arranged to determine whether the name of the function is located in at least the export portions of the one or more executable libraries 465 that have been loaded as part of the runtime environment. For example, the function locator 425 may retrieve a list of dynamic link libraries that have been loaded to implement the runtime environment and scan the export tables of these libraries to determine whether they contain the resolved function name. These dynamic link libraries may comprise system libraries, e.g. libraries that are supplied with the runtime environment and pre-registered with the runtime environment, and/or user-defined libraries, e.g. libraries that have been registered with the runtime environment. A combination of one or more of these library types are referred to herein as executable libraries. In one case registration may occur during an installation process. It may comprise making an entry in a system registry and/or adding a file system location to a PATH variable. In the latter case, all libraries within said file system location may be taken as registered with the runtime environment when the runtime environment is loaded on the computer system. In one case, one or more operating system libraries are automatically registered on installation of the runtime environment. In the case of a user-defined library, the second set of executable code may be loaded as part of the runtime environment independently of a set of system libraries used to implement the runtime environment, in this case the second set of executable code is effectively added as additional user-defined or user-loaded library.

If the function locator 425 locates the name of the function in an export portion of a particular executable library it passes this information, e.g. the name of the located function and the executable library name, to the code generator 435. On receipt of the passed information, the code generator 435 dynamically generates computer program code 445 comprising a platform invoke statement and stores this code in memory 480. The computer program code 445 may be in a modern language that can be compiled to at least intermediate language code such as C# or Java® or in a preferred case may be generated at an intermediate code level, e.g. added to assembled CIL. The latter is preferred as it results in more efficient code and avoids class name pollution. The platform invoke statement references at least the executable library name passed to the code generator 435 and the name of the located function. In one example, the function call detector 415 also forwards to the code generator 435 resolved parameter information derived from the call to the function call detector 415. This resolved parameter information may also be used in the platform invoke statement. For example, the resolved parameter information may indicate variable types and parameter names for each parameter required by the function. The code generator 435 may generate the computer program code 445 by looking up a template string for a platform invoke statement and populating it with information received from the other components at runtime. As well as a platform invoke statement, computer program code may also be generated to use said statement and/or to pass parameters received from a native process (i.e. a second set of executable code) to managed environment variables or objects. In one case, code to use the platform code is generated inline wherever a “CALL” symbol is used in the source code. An exemplary portion of code that is representative of code generated by the code generator 435 is set out below (in an assembled version this may be in a non-human readable form):

.method public static pinvokeimpl ("[located library name]" [received call-convention parameter] lasterr) int32 '[located function name]'([parameter1 type], [parameter2 type], ...[parameterN type]) preservesig { .... }

Once the code generator 435 has created the computer program code 445 in memory 480, the code implementer 455 is notified. The code implementer 455 retrieves the computer program code 445 from memory 480 and prepares it as a third set of executable code. This may comprise a compilation stage, an assembly stage or a formatting stage. The third set of executable code may be native or machine code for the computer system comprising the processor, or it may be intermediate language code such as assembled CIL or Java bytecode that can be implemented by a just-in-time compiler forming part of the runtime environment (e.g. following compilation the intermediate language code is ready for execution by way of the just-in-time compiler). In any case, the code implementer 455 arranges for the third set of executable code to be dispatched to the processor 490 for execution. This may be achieved by calling the method defined by the computer program code 445 with the parameter values received by the function call detector 415. Execution of the third set of executable code by the processor 490 then executes the platform invoke statement. This in turn causes the processor 490 to load and execute the second set of executable code that implements the function, the function being able to use the parameter values provided by the execution of the first set of executable code.

The system components shown in Figure 4 operate at runtime speed, i.e. when code representing the original "CALL" command statement in the first set of executable code is being executed by the processor 490. The present invention provides a "glue" that enables external functions to be implemented even if the function name and/or parameter values are only known at the time of execution. The interoperability bridge unit forms part of a runtime environment and does not require modification of computer program code representing either of the first or second set of executable code. This means that legacy code, and even compiled versions of said code, need not be modified yet may still be linked. This greatly reduces the risk of error, as any modification may lead to an unknown change in function. This is possible, at least partly, by generating additional computer program on-the-fly and then by compiling and executing this code as part of the execution of the first set of executable code. Externally from the computer system the process is transparent, when the execution of the first set of executable code ends, memory may be cleared and the third set of executable code may be deleted.

Method of Compiling Source Code

Figure 5 shows a method 500 of compiling source code. It may be thought of as a method of preparing a computer program for deployment, as it assumes the computer program is complete and defined by the high-level system-independent language; the method then readies this program for execution of a particular computer system. The method 510 is configured to be performed by a compiler unit, such as compiler unit 230 in Figure 2.

The method begins at step 510 with at least a portion of source code 510. This may comprise one of the examples in Figures 3A to 3C. At step 515 the source code 510 is parsed to find one or more “CALL” command statements. This step may comprise scanning an input source code file for a particular string or character array that matches at least “CALL”. At step 520, if one or more “CALL” command statements have been found, a first “CALL” command statement in the source code 510 is analysed. If no “CALL” command statements are located then the method steps are skipped and compilation to intermediate code occurs without adaptation. At step 520, the analysis first determines a target of the “CALL” command statement. This may comprise extracting a string or character array following the “CALL” command statement. This string or character array may comprise the name of a function to be initiated by the “CALL” command statement (i.e. a literal) or the name of a variable storing such a name. In certain cases, a call-convention parameter may directly follow the “CALL” command statement. In this case, the analysis at step 520 may extract the first string or character array following the “CALL” command statement that does not match a list of one or more predefined call-convention parameter values. These values may be set out in a look-up table. These steps may be performed by a parser component of the compiler unit.

At step 525, at attempt is made to locate the determined target of the “CALL” command statement. For example, this may be performed by a function locator component of the compiler unit that is communicatively coupled to the parser component. If the target comprises the name of a variable configured to store a function name, for example as shown in lines 306 of Figures 3B and 3C, then it is determined that the target cannot be located at compile time and the method branches to step 535. If the target comprises the name of a function, e.g. a string or character array within a set of one or more inverted commas as shown in line 306 of Figure 3A, then further steps are initiated to attempt to locate the determined target. This may comprise determining whether the target is present within the source code 510. For example, a compiler unit may store a list of declared internal function names, i.e. names declared and used within the source code 510 itself. At step 525, it may be determined whether the determined target matches a declared internal function name. If it does then the remaining method steps are skipped and compilation to intermediate code occurs without adaptation. If the determined target does not match any of the declared internal function names, i.e. if the "CALL" command statement references a function that is not implemented by the source code 510, then an attempt is made to locate the function in an external source. In one case, one or more executable libraries may be identified in a compiler unit command parameter. For example, a command line compiler unit command may feature the names of one or more dynamic link library files (e.g. "test.dll", "systemlib1.dll") as parameters. In other cases a file comprising a list of executable libraries may be retrieved by the compiler unit. At step 525, to locate a function in an external source, an export portion of one or more of the identified executable libraries may be parsed by the compiler unit to determine if the target is implemented by one of said executable libraries. For example, this may comprise reading an export table of a dynamic link library file and seeing if the determined target matches any entries within said table (e.g. a string match). If a match is found the method continues to step 530 (i.e. the target is located). If a match is not found then step 535 may be performed. In other cases, depending on the particular implementation, if a match is not found at this stage an exception may be initiated.

If the method 500 branches to step 530, the compiler unit generates computer program code comprising a platform invoke statement. This may be performed by a code generator component of the compiler unit. The platform invoke statement references the executable library associated with a located target and the function name associated with the target, e.g. the literal extracted by the parser component. As such the code generator component of the compiler unit and step 530 corresponds to the runtime function of the code generator 435 of Figure 4. The computer program code generated at this step may be similarly represented by the exemplary code set out above with reference to Figure 4. The generated code may include a reference to the source code 510.

If the method 500 branches to step 535, then the compiler unit generates an alternate set of computer program code. This may also be generated by a code generator component of the compiler unit. This alternate set of computer program code comprises a reference to a component of a runtime environment, such as the runtime environment described with reference to Figure 4. For example, it may comprise a call to a predefined function provided by a particular executable library that implements the runtime environment, e.g. the function call detector 415 of Figure 4. This call may be common for all "CALL" command statements for which step 535 applies, e.g. regardless of any variable names for the function and/or independent of the number and type of parameters to be passed to the function.

At step 540 a check is made to see whether all located "CALL" command statements have been processed. If source code 510 comprises a plurality of "CALL" command statements at least steps 520 to 540 may be repeated for each located statement. Once all "CALL" command statements have been processed the method 500 continues to step 545. At this step the compiler unit compiles the source code 510. In one case, the compilation may occur together with any computer program code generated at steps 530 or 535; in another case, the computer program code generated at steps 530 or 535 may comprise executable code, in which case it is simply added to the compiled source code. In any case, legacy code in a first programming language represented by the source code 510 is compiled. Step 545 results in intermediate code 550. The intermediate code 550 may comprise CIL assembly or Java bytecode and as such be executable by a virtual machine forming part of a runtime environment (e.g. a Java virtual machine or common language runtime). In certain cases the intermediate code may be compiled further to produce native or machine code; this may occur for cases that do not use a virtual machine. The output of the method 500 may comprise the first set of executable code 450 of Figure 4. This may be packaged for deployment on the computing system implementing the compiler unit or another computing system.

For example, executable code may be packaged upon and/or copied to a computer-readable storage medium that is then accessed by a further computer device. In one case, executable code may be communicated over a network and stored within a storage device of the further computer device (e.g. stored on a hard disk or flash-memory drive). In use, the executable code may then be retrieved from storage and loaded into memory ready for execution by one or more processors.

In cases where all "CALL" command statements reference a literal, e.g. cases similar to Figure 3A, the output of method 500 may be deployed on a computing system without an interoperability bridge unit. In this case, all adaptation may be performed at compile time. Even if an interoperability bridge unit is not explicitly required, a runtime environment is provided that corresponds to the compiler unit. This runtime environment may still comprise the interoperability bridge unit, which may still be active, i.e. ready to receive a function call if executed code did contain an appropriate reference. Additionally or separately to the above process, compilation may insert other function calls to libraries that form part of the runtime environment. As such, a specific runtime environment may be required to execute executable code generated by the described examples; in this case, the executable code would not run on a different runtime environment. In the case that any "CALL" command statements reference a dynamic variable and/or a target is not located at compile time, an interoperability bridge unit is explicitly required.

Method of Operating an Interoperability Bridge Unit

Figure 6 shows a method 600 that may be implemented by the interoperability bridge unit of Figure 4. At step 610, a first set of executable code is executed within a runtime environment by way of a processor and a memory of a computer system. The first set of executable code may comprise the output of method 500. At step 615, during execution of the first set of executable code, a reference to a function instructed in an original set of source code is received. For example, this may comprise a call to a component of the runtime environment represented in a set of currently-processed executable code. The reference may comprise the reference inserted by step 535 in Figure 5. The reference may be received by the function call detector 415 of Figure 4. At step 620, a target in the form of a function name in the reference is resolved and located. This may be performed by the function locator 425 of Figure 4. Step 620 may comprise determining whether the name of the function is located in at least the export portions of one or more executable libraries that have been loaded into memory by the runtime environment to support execution of executable code. At step 625, it is determined whether the target has been located, e.g. in one of said export portions. If the target is located then step 630 is performed. This involves dynamically generating code in memory that initiates the target. This may be performed by the code generator 435 of Figure 4. At step 635, this generated code is implemented, i.e. it is converted into executable code and dispatched to the processor. Step 635 may be performed by the code implementer 455 of Figure 4. This then implements a second set of executable code to implement function referenced by the target. If at step 625 no target is located then an exception is thrown at step 640. This may be caught by the first set of executable code and/or the runtime environment. For example, in the original source code the "CALL" command statement may comprise an "on exception" clause that is implemented if an exception is thrown. At step 645 the method 600 ends for a particular implementation of a "CALL" command statement. As the first set of executable code is processed further implementations of the same or different "CALL" command statements may occur, in which case method 600 is repeated.

Certain examples as described herein enable different portions of executable code to be "glued" together without modifying original source code for the calling executable code or compiled code for the called code. This enables an application program written in source code to be easily deployed in a variety of different managed environments. For example, in a Microsoft .NET managed environment the compiler and/or interoperability bridge unit may be arranged to perform the linking of executable code within this environment. If an application program needs to be ported to a Java managed environment, the same source code and called code can be used; in this case an implementation of the compiler and/or interoperability bridge unit are configured for Java. In a case where the called code represents an implementation of an operating system function, e.g. display of a message box, this functionality can be provided from the legacy code (i.e. the source code) without modification of this code and using standard operating system libraries. Similar advantages apply if function from system libraries of modern programming languages (e.g. C, C++, C#) are called from legacy code. When migrating legacy code, portions that should not or cannot be modified may be implemented as native code, e.g. using legacy binary executables, and called transparently from unmodified source code that has been compiled within a managed environment.

The above embodiments are to be understood as illustrative examples of the invention. Further embodiments of the invention are envisaged. For example, the location of a target, i.e. a function name received in a reference from executed code, may involve a multi-step process. As a first step, a list of executable code sets loaded into memory may be parsed for the target. For example, this list may comprise all programs that are currently being executed at a particular time by the runtime environment. The list may comprise a list of one or more of: program names, declared method names and declared class names; all of these being associated with loaded executable code. If the target is not found in the first step, a list of classes associated with the system-independent programming language that are logged by the runtime environment may be parsed for the target. If the target is not found in the second step, a name of a executable library that is embedded in the function name may be extracted and used to locate the target. For example, if the runtime value of *function_name* as used in line 306 of Figures 3B and 3C was in the form "[system_lib_name|function_name]" then "system_lib_name" may be used to identify the executable library required for code generation. If the target is not found in the third step, a list of assemblies, i.e. collections of executable libraries already loaded by the runtime environment may be parsed for the target. If the target is not found in the fourth step then a computer-readable medium such as a hard disk may be scanned for a executable library or assembly matching the target.

Another variation of the described examples involves reserving a fixed portion of memory for passing parameters associated with a "CALL" command statement. In one case, computer program code generated by either the code generator 435 or a code generator of a compiler unit also comprises code to "pin" memory, i.e. to reserve a fixed number of memory locations for native parameters. In certain cases, where this is not performed, garbage collection performed by components of the runtime environment may delete parameter data that needs to be passed to a second set of executable code.

Memory may be reserved by generating intermediate code before intermediate code representative of a platform invoke statement that calls a memory pinning function to reserve the memory before a call in the intermediate code associated with the platform invoke statement is implemented. Intermediate code to "unpin", i.e. remove the memory reservation may then be inserted after the call in the intermediate code associated with the platform invoke statement. Memory pinning may be applied where the parameter types used by the "CALL" command statement are not natively supported in the managed environment, e.g. are legacy variable types that do not have an equivalent for compilation to CIL or Java bytecode. For example, when a parameter is passed to a native piece of code, some managed environments may allow portions of memory to be moved and/or rearranged, for example when garbage collection occurs. This typically results in problems for larger data structures in memory or byte arrays. In this case, a runtime environment automatically pins memory to ensure a memory address for a structure or byte array is always consistent for the lifespan of a "CALL" command statement. This is important when working with legacy native code that is unaware that it is being used, i.e. called, from a managed environment.

Certain examples described herein refer to a "CALL" command statement. This is a command statement, i.e. a string that is mapped to a particular set of one or more machine instructions, for implementing a subroutine specified by a function name, i.e. another set of one or more machine instructions that do not sequentially follow the command statement. A "CALL" command statement may have one of three forms: CALL [literal], which can be resolved at compile time; CALL [function_name], where function_name is a variable that needs to be resolved at runtime; or CALL [ptr], where ptr is a pointer to a function name that needs to be resolved at runtime. A "CALL" command statement can take call-convention parameter (which may include the function name) and zero or more parameters. A clause may be provided that determines if a parameter is passed by value, by reference or by content. The first case may just provide a parameter into a function whereas a second case may also provide an output value of a parameter. Parameters are typically variables that are declared within the source code.

Certain examples also refer to a runtime environment. This is sometimes alternatively referred to as a runtime system or runtime. It may generally be considered as a control system to implement a programming language. In certain cases it comprises a virtual machine for implementing bytecode or equivalent. It refers to system components that are present and usable at the time when executable code is loaded into memory to be executed by a processor, i.e. the time at which a computer program is run. The system components provide system resources, i.e. resources at an operating system level or below, to enable the computer system comprising the processor to operate as a programmed machine. System resources may comprise the management of threads (in single or multi-threaded environments), error and exit handling, file handling and file access and/or virtual machine code execution. The system components used by a runtime environment may comprise so-called system libraries. These are packages of executable code to provide the system resources. For example, they may comprise dynamic link libraries or associated assemblies in a Microsoft Windows environment or .jar files in a Java environment. A set of one or more system libraries may comprise core runtime environment files and/or may comprise executable code relating to other application programs that is imported into the run-time environment, e.g. user libraries that are registered with the runtime environment. For example, in a case wherein the second set of executable code is directly compiled from computer program code written in the system-independent programming language, this second set of executable code may be loaded into the runtime environment as a user-defined library for access by a function locator. Where reference is made to called or native functions, i.e. functions referenced by a “CALL” command statement, these may relate to a third party application program interface or a further native program.

In certain cases, a second compiler unit 235 may be provided separately, e.g. may form part of an operating system and/or be installed separately. For example, a common language runtime or a Java virtual machine may already be present on a computer system or may be provided by way of independent installation processes.

In one variation of a computer system, a compile command receiver may receive an identified executable library as a parameter. This variation may apply to the method. In one variation of a method the reference to the function received from the first set of executable code comprises a function call type, the name of the function and data indicative of one or more parameters required by the function, and the step of dynamically generating computer program code comprises generating computer program code comprising a platform invoke statement referencing the particular executable library, the name of the function and one or more parameter types as determined from the data indicative of one or more parameters required by the function. Any described example may add or omit features from other described examples.

The term “deployment” is used herein to describe the process of placing an application program in a suitable form for execution by one or more processors. Deployment may comprise an installation process that copies object or intermediate code to a storage device of a computer system and that configures said code for execution. This configuration may comprise registering the object or intermediate code with an operating system and/or a runtime environment. The phrase “preparing for deployment” is used to describe a compilation process that operates upon source code to produce object or intermediate code suitable for execution. In one case, both preparing for deployment and deployment may form part of a common process. The term “legacy” is used herein to describe features that relate to a computing architecture that is no longer in use. For example, a legacy programming language is a programming language that was configured to operate on a superseded (i.e. legacy) computing architecture. In certain case, the legacy programming language may relate to a specification that has itself been superseded. Even though computer systems have been described as including one or more processors and a memory, these components may be excluded in particular cases.

As set out herein certain examples provide an interoperability bridge unit arranged to, when a first set of executable code is implemented in a runtime environment, implement a second set of executable code for the computer system in the runtime environment to provide a function instructed in the system-independent programming language. The interoperability bridge may form part of the runtime environment.

It is to be understood that any feature described in relation to any one embodiment may be used alone, or in combination with other features described, and may also be used in combination with one or more features of any other of the embodiments, or any combination of any other of the embodiments. Furthermore, equivalents and modifications not described above may also be employed without departing from the scope of the invention, which is defined in the accompanying claims.

Claims

1. A computer system comprising:

at least one processor;

a memory coupled to the at least one processor;

a computer-readable storage medium comprising a first set of executable code representative of an application program, wherein the first set of executable code is derived from intermediate language code, the intermediate language code being compiled from computer program code written in a system-independent programming language;

a runtime environment arranged to support execution of the first set of executable code by way of the at least one processor and the memory, the runtime environment comprising:

one or more executable libraries registered with the runtime environment, the one or more executable libraries comprising executable code to implement at least the runtime environment on the computer system, each executable library comprising an export portion listing any functions implemented by the executable code;

an interoperability bridge unit arranged to, when the first set of executable code is implemented in the runtime environment, implement a second set of executable code for the computer system in the runtime environment to provide a function instructed in the system-independent programming language, the interoperability bridge unit comprising:

a function call detector arranged to receive a reference to the function from the first set of executable code;

a function locator arranged to resolve the reference to determine at least a name of the function, the function locator being further arranged to determine whether the name of the function is located in at least the export portions of the one or more executable libraries;

a code generator arranged to, responsive to the name of the function being located in an export portion of a particular executable library, dynamically generate, in the memory, computer program code comprising a platform invoke statement, the platform invoke statement referencing at least the particular executable library and the name of the function; and

a code implementer to prepare the generated computer program code as a third set of executable code and dispatch said third set of executable code for execution by the processor, wherein execution of the third set of executable code by the processor executes the second set of executable code.

2. A computer system according to claim 1, wherein the one or more executable libraries are one or more of system libraries and user-loaded libraries, the one or more executable libraries comprising one or more dynamic link libraries and wherein the export portion of each executable library comprises one or more declared export statements.

3. A computer system according to claim 1 or 2, wherein the intermediate language code comprises one of:

code in a common intermediate language as defined by a common language infrastructure specification; and

Java bytecode.

4. A computer system according to any one of claims 1 to 3,

wherein the reference to the function received from the first set of executable code comprises a function call type, the name of the function and data indicative of one or more parameters required by the function, and

wherein the code generator is arranged to generate computer program code comprising a platform invoke statement referencing the particular executable library, the name of the function and one or more parameter types as determined from the data indicative of one or more parameters required by the function.

5. A computer system comprising:

at least one processor;

a memory coupled to the at least one processor;

a compiler unit arranged to compile computer program code written in a system-independent programming language to intermediate language code, said intermediate language code being executable for a defined computing system, the compiler unit comprising:

a parser arranged to parse the computer program code and determine if the computer program code comprises at least one function call command statement that references a function that is not implemented by the computer program code;

a function locator communicatively coupled to the parser to, responsive to a positive determination from the parser, resolve a function call command statement to determine at least a function name, the function locator being further arranged to determine whether the function name is located in an export portion of an identified executable library; and

a code generator communicatively coupled to the function locator to, responsive to the location of a function name in an export portion of an identified executable library by the function locator, generate computer program code, the computer program code comprising a platform invoke statement, the platform invoke statement referencing the identified executable library and the function name,

wherein the compiler unit is arranged to compile the computer program code written in a system-independent programming language to intermediate language code and to prepare said intermediate language code together with the computer program code generated by the code generator as executable code for the defined computing system.

6. A computer system according to claim 5, wherein the one or more executable libraries comprise one or more dynamic link libraries and wherein the export portion of each executable library comprises one or more declared export statements.

7. A computer system according to claim 5 or 6, wherein the intermediate language code comprises one of:

code in a common intermediate language as defined by a common language infrastructure specification; and

Java bytecode.

8. A computer system according to any one of claims 5 to 7, wherein the code generator is arranged to generate computer program code that reserves a fixed portion of the memory for one or more parameters referenced by the function call command statement.

9. A computer system according to any one of claims 5 to 8, wherein the compiler unit comprises:

a compile command receiver to receive the identified executable library as a parameter.

10. A computer-implemented method of deploying an application program into a runtime environment of a computing system, the computing system comprising at least one processor coupled to a memory, the method comprising:

retrieving, from a computer-readable storage medium coupled to the computing system, a first set of executable code for the computing system representative of an application program, wherein said first set of executable code is compiled from intermediate language code, the intermediate language code being compiled from computer program code written in a system-independent programming language;

executing the first set of executable code in the runtime environment by way of the at least one processor and the memory, the runtime environment comprising one or more executable libraries comprising executable code to implement at least the runtime environment on the computing system, each executable library comprising an export portion listing any functions implemented by the executable code;

receiving a reference to a function instructed in the system-independent programming language, the function being implemented by second set of executable code, the second set of executable code being separate from the first set of executable code;

resolving the reference to determine at least a name of the function;

determining whether the name of the function is located in at least the export portions of the one or more executable libraries;

responsive to the name of the function being located in an export portion of a particular executable library, dynamically generating, in the memory, computer program code comprising a platform invoke statement, the platform invoke statement referencing the particular executable library and the name of the function;

preparing the generated computer program code as a third set of executable code; and

executing the third set of executable code in the runtime environment by way of the at least one processor and the memory, wherein execution of the third set of executable code by the at least one processor executes the second set of executable code and implements the function.

11. A computer-implemented method according to claim 10, wherein the one or more executable libraries are one or more of system libraries and user-loaded libraries, the one or more executable libraries comprising one or more dynamic link libraries and wherein the export portion of each executable library comprises one or more declared export statements.

12. A computer-implemented method according to claim 10 or 11, wherein the intermediate language code comprises one of:

code in a common intermediate language as defined by a common language infrastructure specification; and

Java bytecode.

13. A computer-implemented method according to any one of claims 10 to 12,

wherein the reference to the function received from the first set of executable code comprises a function call type, the name of the function and data indicative of one or more parameters required by the function, and

wherein the step of dynamically generating computer program code comprises generating computer program code comprising a platform invoke statement referencing the particular executable library, the name of the function and one or more parameter types as determined from the data indicative of one or more parameters required by the function.

14. A computer-implemented method of preparing an application program for deployment into a runtime environment of a computing system, the computing system comprising at least one processor coupled to a memory, the method comprising:

retrieving, from a computer-readable storage medium, a first set of computer program code representative of the application program, the first set of computer program code being written in a system-independent programming language;

parsing the computer program code to determine if the first set of computer program code comprises a function call command statement that references a function that is not implemented by the first set of computer program code;

responsive to a determination that the first set of computer program code comprises a function call command statement that references a function that is not implemented by the first set of computer program code, resolving the function call command statement to determine at least a function name;

determining whether the function name is located in an export portion of an identified executable library;

responsive to the location of the function name in the export portion, generating a second set of computer program code, the second set of computer program code comprising a platform invoke statement, the platform invoke statement referencing the identified executable library and the function name;

compiling the first and second sets of computer program code to intermediate language code;

preparing the intermediate language code as a first set of executable code for the computing system, wherein execution of the first set of executable code by the at least one processor of the computing system executes a second set of executable code and implements the function.

15. A computer-implemented method according to claim 14, wherein the one or more executable libraries are one or more of system libraries and user-loaded libraries, the one or more executable libraries comprising one or more dynamic link libraries and wherein the export portion of each executable library comprises one or more declared export statements.

16. A computer-implemented method according to claim 14 or 15, wherein the intermediate language code comprises one of:

code in a common intermediate language as defined by a common language infrastructure specification; and

Java bytecode.

17. A computer-implemented method according to any one of claims 14 to 16,

wherein the function call command statement comprises a function call type, the name of the function and data indicative of one or more parameters required by the function, and

wherein the step of generating a second set of computer program code comprises generating a second set of computer program code comprising a platform invoke statement referencing the identified executable library, the name of the function and one or more parameter types as determined from the data indicative of one or more parameters required by the function.

18. A computer-implemented method according to claim 17, wherein the step of generating a second set of computer program code comprises generating computer program code that reserves a fixed portion of the memory for said one or more parameters required by the function.

19. A computer-implemented method according to any one of claims 14 to 16, comprising:

receiving the identified executable library as a parameter.

20. A computer-implemented method of deploying an application program into a runtime environment of a computing system, the computing system comprising at least one processor coupled to a memory, the method comprising:

retrieving, from a computer-readable storage medium, a first set of computer program code representative of the application program, the first set of computer program code being written in a system-independent programming language;

parsing the computer program code to determine if the first set of computer program code comprises a function call command statement that references a function that is not implemented by the first set of computer program code,

wherein, responsive to a determination that the first set of computer program code comprises a function call command statement that references a function that is not implemented by the first set of computer program code, the step of parsing comprises:

resolving the function call command statement to determine at least a function name; and

determining whether the function name comprises a static or dynamic variable,

wherein, responsive to the function name comprising a static variable, the step of determining comprises:

determining whether the function name is located in an export portion of an identified executable library; and

responsive to the location of the function name in the export portion, generating a second set of computer program code, the second set of computer program code comprising a platform invoke statement, the platform invoke statement referencing the identified executable library and the function name; and

wherein, responsive to the function name comprising a dynamic variable, the step of determining comprises generating a second set of computer program code, the second set of computer program code comprising a runtime reference to the function;

compiling the first and second sets of computer program code to intermediate language code;

preparing the intermediate language code as a first set of executable code for the computing system;

executing the first set of executable code in the runtime environment by way of the at least one processor and the memory, the runtime environment comprising one or more executable libraries comprising executable code to implement at least the runtime environment on the computing system, each executable library comprising an export portion listing any functions implemented by the executable code,

wherein, responsive to receipt of the runtime reference, the step of executing comprises:

resolving the runtime reference to determine at least a name of the function; and

determining whether the name of the function is located in at least the export portions of the one or more executable libraries,

wherein, responsive to the name of the function being located in an export portion of a particular executable library, the step of determining comprises:

dynamically generating, in the memory, computer program code comprising a platform invoke statement, the platform invoke statement referencing the particular executable library and the name of the function;

preparing the generated computer program code as a third set of executable code; and

executing the third set of executable code in the runtime environment by way of the at least one processor and the memory; and

wherein execution, by way of the at least one processor, of the first set of executable code for a static variable or the third set of executable code for a dynamic variable executes a second set of executable code and implements the function.

21. A computer program comprising computer program code arranged to, when loaded into system memory and processed by one or more processors, causes said processers to perform the computer-implemented methods of any one of claims 10 to 20.

22. A computer-readable storage medium having recorded thereon the computer program of claim 21.

23. A computer system, substantially as hereinbefore described with reference to the accompanying drawings.

24. A computer-implemented method substantially as hereinbefore described with reference to the accompanying drawings.

Amendment to the claims have been incorporated as follows

Claims

1. A computer system comprising:

at least one processor;

a memory coupled to the at least one processor;

a computer-readable storage medium comprising a first set of executable code representative of an application program, wherein the first set of executable code is derived from intermediate language code, the intermediate language code being compiled from computer program code written in a system-independent programming language;

a runtime environment arranged to support execution of the first set of executable code by way of the at least one processor and the memory, the runtime environment comprising:

one or more executable libraries registered with the runtime environment, the one or more executable libraries comprising executable code to implement at least the runtime environment on the computer system, each executable library comprising an export portion listing any functions implemented by the executable code;

an interoperability bridge unit arranged to, when the first set of executable code is implemented in the runtime environment, implement a second set of executable code for the computer system in the runtime environment to provide a function instructed in the system-independent programming language, the interoperability bridge unit comprising:

a function call detector arranged to receive a reference to the function from the first set of executable code;

a function locator arranged to resolve the reference to determine at least a name of the function, the function locator being further arranged to determine whether the name of the function is located in at least the export portions of the one or more executable libraries;

a code generator arranged to, responsive to the name of the function being located in an export portion of a particular executable library, dynamically generate, in the memory, computer program code comprising a platform invoke statement, the platform invoke statement referencing at least the particular executable library and the name of the function; and

a code implementer to prepare the generated computer program code as a third set of executable code and dispatch said third set of executable code for execution by the processor, wherein execution of the third set of executable code by the processor executes the second set of executable code.

2. A computer system according to claim 1, wherein the one or more executable libraries are one or more of system libraries and user-loaded libraries, the one or more executable libraries comprising one or more dynamic link libraries and wherein the export portion of each executable library comprises one or more declared export statements.

3. A computer system according to claim 1 or 2, wherein the intermediate language code comprises one of:

code in a common intermediate language as defined by a common language infrastructure specification; and

Java bytecode.

4. A computer system according to any one of claims 1 to 3,

wherein the reference to the function received from the first set of executable code comprises a function call type, the name of the function and data indicative of one or more parameters required by the function, and

wherein the code generator is arranged to generate computer program code comprising a platform invoke statement referencing the particular executable library, the name of the function and one or more parameter types as determined from the data indicative of one or more parameters required by the function.

5. A computer-implemented method of deploying an application program into a runtime environment of a computing system, the computing system comprising at least one processor coupled to a memory, the method comprising:

retrieving, from a computer-readable storage medium coupled to the computing system, a first set of executable code for the computing system representative of an application program, wherein said first set of executable code is compiled from intermediate language code, the intermediate language code being compiled from computer program code written in a system-independent programming language;

executing the first set of executable code in the runtime environment by way of the at least one processor and the memory, the runtime environment comprising one or more executable libraries registered with the runtime environment and comprising executable code to implement at least the runtime environment on the computing system, each executable library comprising an export portion listing any functions implemented by the executable code;

receiving a reference to a function from the first set of executable code, the function being instructed in the system-independent programming language and being implemented by second set of executable code, the second set of executable code being separate from the first set of executable code;

resolving the reference to determine at least a name of the function;

determining whether the name of the function is located in at least the export portions of the one or more executable libraries;

responsive to the name of the function being located in an export portion of a particular executable library, dynamically generating, in the memory, computer program code comprising a platform invoke statement, the platform invoke statement referencing the particular executable library and the name of the function;

preparing the generated computer program code as a third set of executable code; and

executing the third set of executable code in the runtime environment by way of the at least one processor and the memory, wherein execution of the third set of executable code by the at least one processor executes the second set of executable code and implements the function.

6. A computer-implemented method according to claim 5, wherein the one or more executable libraries are one or more of system libraries and user-loaded libraries, the one or more executable libraries comprising one or more dynamic link libraries and wherein the export portion of each executable library comprises one or more declared export statements.

7. A computer-implemented method according to claim 5 or 6, wherein the intermediate language code comprises one of:

code in a common intermediate language as defined by a common language infrastructure specification; and

Java bytecode.

8. A computer-implemented method according to any one of claims 5 to 7,

wherein the reference to the function received from the first set of executable code comprises a function call type, the name of the function and data indicative of one or more parameters required by the function, and

wherein the step of dynamically generating computer program code comprises generating computer program code comprising a platform invoke statement referencing the particular executable library, the name of the function and one or more parameter types as determined from the data indicative of one or more parameters required by the function.

9. A computer-implemented method of deploying an application program into a runtime environment of a computing system, the computing system comprising at least one processor coupled to a memory, the method comprising:

retrieving, from a computer-readable storage medium, a first set of computer program code representative of the application program, the first set of computer program code being written in a system-independent programming language;

parsing the computer program code to determine if the first set of computer program code comprises a function call command statement that references a function that is not implemented by the first set of computer program code, wherein, responsive to a determination that the first set of computer program code comprises a function call command statement that references a function that is not implemented by the first set of computer program code, the step of parsing comprises:

resolving the function call command statement to determine at least a function name; and

determining whether the function name comprises a static or dynamic variable,

wherein, responsive to the function name comprising a static variable, the step of determining comprises:

determining whether the function name is located in an export portion of an identified executable library; and

responsive to the location of the function name in the export portion, generating a second set of computer program code, the second set of computer program code comprising a platform invoke statement, the platform invoke statement referencing the identified executable library and the function name; and

wherein, responsive to the function name comprising a dynamic variable, the step of determining comprises generating a second set of computer program code, the second set of computer program code comprising a runtime reference to the function;

compiling the first and second sets of computer program code to intermediate language code;

preparing the intermediate language code as a first set of executable code for the computing system;

executing the first set of executable code in the runtime environment by way of the at least one processor and the memory, the runtime environment comprising one or more executable libraries registered with the runtime environment and comprising executable code to implement at least the runtime environment on the computing system, each executable library comprising an export portion listing any functions implemented by the executable code,

wherein, responsive to receipt of the runtime reference from the first set of executable code, the step of executing comprises:

resolving the runtime reference to determine at least a name of the function; and

determining whether the name of the function is located in at least the export portions of the one or more executable libraries,

wherein, responsive to the name of the function being located in an export portion of a particular executable library, the step of determining comprises:

dynamically generating, in the memory, computer program code comprising a platform invoke statement, the platform invoke statement referencing the particular executable library and the name of the function;

preparing the generated computer program code as a third set of executable code; and

executing the third set of executable code in the runtime environment by way of the at least one processor and the memory; and

wherein execution, by way of the at least one processor, of the first set of executable code for a static variable or the third set of executable code for a dynamic variable executes a second set of executable code and implements the function.

19 11 13

10. A computer program comprising computer program code arranged to, when loaded into system memory and processed by one or more processors, causes said processers to perform the computer-implemented methods of any one of claims 5 to 9.

11. A computer-readable storage medium having recorded thereon the computer program of claim 10.

12. A computer system, substantially as hereinbefore described with reference to the accompanying drawings.

13. A computer-implemented method substantially as hereinbefore described with reference to the accompanying drawings.

Application No:	GB1314586.7	**Examiner:**	Mr Jonathan Golding
Claims searched:	1 - 24	**Date of search:**	25 September 2013

Patents Act 1977: Search Report under Section 17

Documents considered to be relevant:

Category	Relevant to claims	Identity of document and passage or figure of particular relevance
A	-	US 2006/0225033 A1 (YE ET AL)
A	-	US 2005/0172299 A1 (ZHAO ET AL)
A	-	US 2003/0182460 A1 (KHARE)

Categories:

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

Field of Search:

Search of GB, EP, WO & US patent documents classified in the following areas of the UKC^{X} :

Worldwide search of patent documents classified in the following areas of the IPC

G06F

The following online and other databases have been used in the preparation of this search report

EPODOC & WPI; TXTE; XPI3E, XPIPCOM and XPLNCS; Internet.

International Classification:

Subclass	Subgroup	Valid From
G06F	0009/445	01/01/2006
G06F	0009/45	01/01/2006