Methods and systems for optimizing a build order of component source modules comprises creating a dependency graph based on dependency information. Historical build information associated with previous build failures is then used to calculate relative failure factors for paths of the dependency graph, and the relative failure factors are used to determine an order of traversal of the dependency graph during a build process in which component binary modules are built from the component source modules.
Network 16 (e.g., Internet)

Server 14

Build Service 20

Source Code Management Repository 22

Component Source Modules 26

Binary Repository 24

Component Binary & Application Modules 28

Historical Build Information 30

Hosting System 12

FIG. 1
Create a dependency graph based on dependency information

Use historical build information associated with previous build failures to calculate relative failure factors for paths of the dependency graph

Use the relative failure factors to determine an order of traversal of the dependency graph during a build process in which component binary and application modules are built from the component source modules

FIG. 2
Determining for each of the path options in the global reverse dependency graph, a failure factor value Q

Determine whether a build should be optimized for failure first or failure last

Has a node been reached having at least two path options, but dependencies do not determine which path for the build process to take?

If the build is optimized for failure first, then traverse the global reverse dependency graph and perform the build based on the path option having a highest failure factor value

If the build is optimized for failure last, then traverse the global reverse dependency graph and perform the build based on the path option having a lowest failure factor value

FIG. 4
USING BUILD HISTORY INFORMATION TO OPTIMIZE A SOFTWARE BUILD PROCESS

BACKGROUND OF THE INVENTION

There are hosting systems in which developers may upload their source code modules for storage in a source code management (SCM) repository. Examples of such SCM repositories include CVS™ and Subversion™. The SCM repository, which contains source code modules, is used to build binary libraries and application modules. The binary libraries and application modules can be stored in a binary repository. Examples of such binary repositories include Redhat Network™, Yum™, Maven™, and CPAN™ for example.

The source code modules are compiled and linked into the binary modules during one or more build processes, and the source code modules have defined dependencies that may determine the possible orders in which the build processes can execute. There may be multiple independent build processes that need to run at any given time, and the build processes may need to decide among multiple possible paths of execution. The dependency information becomes particularly important as the number of libraries/applications that are being built is very large, e.g. over 100 libraries, and where the libraries are rapidly changing. In this case, a great deal of processing power is necessary to keep up with the changes and for efficiency, only the source code modules that are changed should be rebuilt.

Depending on the nature of build components, such as the source code modules, and their usefulness as independent components, it might be more efficient to perform the build in such a way as to find failing source code modules first, which is referred to as "failure first", or to find failing source code modules last, which is referred to as "failure last". If the components have usefulness independently, or the build process can start at a node in a dependency graph where the process failed last, then it is more efficient to find failure last, because more components will be published before a failure occurs. Unfortunately, it may not be apparent how to optimize the build order of the component source modules for failure first or failure last based on the dependency information alone.

BRIEF SUMMARY OF THE INVENTION

Methods and systems for optimizing a build order of component source modules comprises creating a dependency graph based on dependency information. Historical build information associated with previous build failures is then used to calculate relative failure factors for paths of the dependency graph; and the relative failure factors are used to determine an order of traversal of the dependency graph during a build process in which component binary modules are built from the component source modules.

BRIEF DESCRIPTION OF SEVERAL VIEWS OF THE DRAWINGS

FIG. 1 is a block diagram illustrating an exemplary network system environment in which one embodiment of the present invention may be implemented for optimizing a build order of component source modules.
source modules 26 when the dependency information alone is insufficient to determine the build order, as explained below. Although the build service 20 is shown as one component, the build process may be implemented as multiple build processes, and the functionality of the build service 20 may be implemented with a greater number of components and run on the same or on multiple servers 14.

FIG. 2 is a flow diagram illustrating a process for optimizing a build order of component source modules 26. In the exemplary embodiment, the process is performed by the build service 20, with or without the aid of additional applications. The process may begin by creating a dependency graph (block 200). In one embodiment, a global reverse dependency graph is created by merging dependency files associated with the component source modules 26 and reversing the directions of the edges in the dependency graph.

In one embodiment the process may begin by polling the source code management repository 22 for any changes in dependency information in the component source modules 26. In one embodiment, both the source code management repository 22 and/or the binary repository 24 may be polled for changes. In one embodiment, polling may be initiated manually whereby the polling is initiated by a developer via a client computer 18. In another embodiment, polling may be initiated automatically. Automatic polling may be triggered by an expiration of a configured time interval. If no changes are detected, then the polling may be rescheduled for a later time. A continuous integration build server, such as Cruise Control or AntHill can be used to poll the source code management repository 22 and/or the binary repository 24 for changes. Alternatively, a time-based scheduling service, such as “cron” of UNIX-like operating systems, could be used. In response to detecting changes in the dependency information, the dependency graph may be created based on the dependency information.

The historical build information 30 associated with previous build failures is used to calculate relative failure factors for paths of the dependency graph (block 202). The relative failure factors of the paths are then used to determine an order of traversal of the dependency graph during a build process in which component binary and application modules 28 are built from the component source modules 26 (block 204).

According to the exemplary embodiment, the historical build information 30 may include the following:

- A history of which respective component source modules 26 have failed most often in the past as a percentage of the total attempts to build the respective component source modules;
- A history of the developer’s contributions to build failures as a percentage of developer’s total contributions;
- A total number of file changes since a last successful build.

In one embodiment, when a component source module 26 is used in a build, historical build data is recorded for the source module and the developer of the source module in the historical build information 30. The list of historical build information may be readily changed and expanded. For example, information such as time of day may be stored to determine whether network access or other outside variables are affecting the build process. These pieces of information can be used to show a relative likelihood of failure.

FIGS. 3A-3D are diagrams illustrating an example creation of a global reverse dependency graph. A global reverse dependency graph is created from a merging of the dependencies from two modules, employee.demo and zero.services.rating, and then a reversal of the directions of the edges of the merged graph. FIG. 3A shows a dependency graph 300 of the dependencies for the employee.demo module. An example xml version of the employee.demo module could be:

```
<ivy-module version="1.3">
  <info module="employee.demo" organisation="Zero">
    <license name="type of license" url=http://license.page/>
    <description homepage=http://module.description.page/>
    <publications></publications>
    <dependencies>
      <dependency name="zero.core" org="Zero" rev="1.0">
    </dependency>
    <dependency name="dojo" org="dojo" rev="0.4.3">
    </dependency>
    <dependency name="data" org="zero" rev="1.0.7">
    </dependency>
  </dependencies>
</ivy-module>
```

FIG. 3B shows a dependency graph 302 of the dependencies for the zero.services.rating model. An example xml version of the employee.demo module could be:

```
<ivy-module version="1.3">
  <info module="zero.services.rating" organisation="Zero">
    <license name="IBM" url=http://www.ibm.com/>
    <description homepage=http://www.projectzero.org/>
    <publications></publications>
    <dependencies>
      <dependency name="zero.core" org="Zero" rev="1.0.7">
    </dependency>
    <dependency name="dojo" org="dojo" rev="0.4.3">
    </dependency>
    <dependency name="derby" org="org.apache.derby" rev="10.2.2.4">
    </dependency>
  </dependencies>
</ivy-module>
```

FIG. 3C shows a merged global dependency graph 304 for the modules employee.demo and zero.services.rating; and FIG. 3D shows a global reverse dependency graph 306, which is created by reversing directions of the edges of the merged global dependency graph 304.

FIG. 4 is a flow diagram illustrating a process for calculating relative failure factors for paths of the dependency graph, described in block 204 of FIG. 2, in further detail. The process may begin by determining for each of the path options in the global reverse dependency graph 306, a failure factor value Q (block 400). According to one exemplary embodiment, Q may be defined as

```
employee.demo/config/ivy.xml:
```

```
<ivy-module version="1.3">
  <info module="employee.demo" organisation="Zero">
    <license name="type of license" url=http://license.page/>
    <description homepage=http://module.description.page/>
    <publications></publications>
    <dependencies>
      <dependency name="zero.core" org="zero" rev="1.0">
    </dependency>
    <dependency name="dojo" org="dojo" rev="0.4.3">
    </dependency>
    <dependency name="data" org="zero" rev="1.0.7">
    </dependency>
  </dependencies>
</ivy-module>
```

```
zero.services.rating/config/ivy.xml:
```

```
<ivy-module version="1.0" encoding="UTF-8">
  <ivy-module version="1.3">
    <info module="zero.services.rating" organisation="Zero">
      <license name="IBM" url=http://www.ibm.com/>
      <description homepage=http://www.projectzero.org/>
      <publications></publications>
      <dependencies>
        <dependency name="zero.core" org="Zero" rev="1.0.7">
        </dependency>
        <dependency name="dojo" org="dojo" rev="0.4.3">
        </dependency>
        <dependency name="derby" org="org.apache.derby" rev="10.2.2.4">
        </dependency>
      </dependencies>
    </dependencies>
</ivy-module>
```

```
<ivy-module version="1.0" encoding="UTF-8">
  <ivy-module version="1.3">
    <info module="employee.demo" organisation="Zero">
      <license name="type of license" url=http://license.page/>
      <description homepage=http://module.description.page/>
      <publications></publications>
      <dependencies>
        <dependency name="zero.core" org="Zero" rev="1.0">
        </dependency>
        <dependency name="dojo" org="dojo" rev="0.4.3">
        </dependency>
        <dependency name="data" org="zero" rev="1.0.7">
        </dependency>
      </dependencies>
    </dependencies>
</ivy-module>
```
where $P_r$ represents a percentage of past failures for a particular component source module; 

$P_d$, represents the percentage of past failure for a particular developer ($d$), who committed code to the particular component source module; and

$F_d$, represents a number of files committed by the particular developer ($d$).

After the failure factor values have been calculated, it is determined whether a build should be optimized for failure first or failure last (block 402). As stated above, failure first refers to performing the build in such a way as to find failing component source modules 26 first, while failure last refers to performing the build in such a way as to find a failing component source modules 26 last. The determination of optimizing a build based on failure first or failure last may be based upon user input.

When traversing the global reverse dependency graph 306 during the build process, it is determined whether a node has been reached having at least two path options, but dependencies do not determine which path for the build process to take (block 404). If so, and if the build is optimized for failure first, then the build service 20 traverses the global reverse dependency graph 306 and performs the build based on the path options having a highest failure factor value (block 406). If the build is optimized for failure last, then the build service 20 traverses the global reverse dependency graph 306 and performs the build based on the path options having a lowest failure factor value (block 408).

During the actual build process, the build service 20 may check the source code management repository 22 to determine whether the component source modules 26 have changed (e.g., using a ‘svn info’ command in the case of a subversion repository), and, whether the changes require compilation or just runtime testing (i.e., skip the building phase). In one embodiment, the component source modules 26 may need recompiling if the compiled source file types for the component have changed, or if any direct compilation dependencies have changed. If any other changes have occurred within a component source module 26, then the component source module 26 only needs to be republished (e.g., config files, scripts). Following the example described in FIGS. 3A-3D and the example XML, the zero.services.rating, zero.data, zero.core, dojo, and derby modules are direct dependencies since they are referenced directly by the ivy.xml module. On the other hand, zero.network and zero.network.support are transitive dependencies.

To determine the ordered list of component source modules 26 that need to be compiled, the build service 20 traverses the global reverse dependency graph 306. For each node in the global reverse dependency graph 306 the build service 20 may build the node if there was a change that requires it, i.e., compiled source file types changed, or a compiled source file types of a directly connected node changed. Using the example global reverse dependency graph 306 above, if a change to a compiled source file type was detected in zero.network.support, then the build service 20 would need to build zero.network.support and zero.network since there could have been API changes in zero.network.support that could affect the compilation of zero.network.

In one embodiment, the build service 20 may publish the build results to a temporary working repository that is used by the hosting system 12 until all changes have been fully verified by completing all of the above defined phases/ steps which ends in publication. Builds may resolve against a chain of binary repositories, where the temporary binary repository has greater precedence than the publicly available binary repository 24. This may allow recently built libraries to be resolved against before the libraries are published.

After the component binary and application modules 28 have been built, the build service 20 may test the component binary and application modules 28 based on change information and the dependency graph. The set of component binary and application modules 28 that will need to be tested will be equal to or greater than the set of component binary and application modules 28 that needed to be built during the build phase, since the set includes a full set of transitive dependencies and because the changes that require testing might not be included in those that require building. Continuing with the example described in FIGS. 3A-3D, if zero.network.support had any change, then any connected node (either transitive or directly) would need to be tested. This would include zero.network, zero.core, zero.data, employee, demo, and zero.services.rating.

After testing, the build service 20 may publish new libraries of component binary and application modules 28 from a temporary working repository to the binary repository 24, and clean the temporary working repository.

A method and system for optimizing a build order of component source module for failure first or failure last has been disclosed. 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, etc.

Furthermore, the invention can take the form of a computer program product accessible from a computer-readable or computer-readable medium providing program code for use by or in connection with a computer or any instruction execution system. For the purposes of this description, a computer-readable or computer-readable medium can be any apparatus that can contain, store, communicate, propagate, or transport the program for use by or in connection with the instruction execution system, apparatus, or device.

The medium can be an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system (or apparatus or device) or a propagation medium. Examples of a computer-readable medium include a semiconductor or solid state memory, magnetic tape, a removable computer diskette, a random access memory (RAM), a read-only memory (ROM), a rigid magnetic disk and an optical disk. Current examples of optical disks include compact disk-read only memory (CD-ROM), compact disk-read/write (CD-R/W) and DVD.

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 stor-
age of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

[0041] 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.

[0042] 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.

[0043] The present invention has been described in accordance with the embodiments shown, and one of ordinary skill in the art will readily recognize that there could be variations to the embodiments, and any variations would be within the spirit and scope of the present invention. Accordingly, many modifications may be made by one of ordinary skill in the art without departing from the spirit and scope of the appended claims.

We claim:

1. A method for optimizing a build order of component source modules, comprising:

   creating a dependency graph based on dependency information;

   using historical build information associated with previous build failures to calculate relative failure factors for paths of the dependency graph;

   using the relative failure factors to determine an order of traversal of the dependency graph during a build process in which component binary modules are built from the component source modules.

2. The method of claim 1 wherein the historical build information includes at least one of:

   a history of which respective component source modules have failed most often as a percentage of total attempts to build the respective component source modules;

   a history of a developer's contributions to build failures as a percentage of a total contribution of the developer;

   a total number of file changes since a last successful build.

3. The method of claim 1 further comprising:

   polling a source code management repository for any changes in the dependency information in the component source modules;

   in response to detecting changes in the dependency information, creating the dependency graph based on the dependency information.

4. The method of claim 1 wherein creating the dependency graph comprises creating a global reverse dependency graph.

5. The method of claim 4 further comprising creating the global reverse dependency graph by reversing directions of edges of a merged global dependency graph.

6. The method of claim 4 further comprising calculating the relative failure factors by determining for each of a plurality of path options in the global reverse dependency graph, a failure factor value.

7. The method of claim 6 wherein the using the relative failure factors further comprises:

   if a build is optimized for failure first, then traversing the global reverse dependency graph and performing the build based on the plurality of path options having a highest failure factor value.

8. The method of claim 6 wherein the using the relative failure factors further comprises:

   if the build is optimized for failure last, then traversing the global reverse dependency graph and performing the build based on the plurality of path options having a lowest failure factor value.

9. The method of claim 6 wherein the failure factor value is designated as Q, where Q is defined as:

   \[ Q = P_r \frac{\sum_{i=0}^{n} (P_{d_i} \cdot F_{d_i})}{\sum_{i=0}^{n} F_{d_i}} \]

   where \( P_r \) represents a percentage of past failures for a particular component source module;

   \( P_{d_i} \) represents the percentage of past failure for a particular developer (\( d_i \)) who committed code to the particular component source module; and

   \( F_{d_i} \) represents a number of files committed by the particular developer (\( d_i \)).

10. A system comprising:

    a server; and

    a build service executing on the server; the build service configured to:

    create a dependency graph based on dependency information;

    use historical build information associated with previous build failures to calculate relative failure factors for paths of the dependency graph; and

    use the relative failure factors to determine an order of traversal of the dependency graph during a build process in which component binary modules are built from the component source modules.

11. The system of claim 10 wherein the historical build information includes at least one of:

    a history of which respective component source modules have failed most often as a percentage of total attempts to build the respective component source modules;

    a history of a developer's contributions to build failures as a percentage of a total contribution of the developer;

    a total number of file changes since a last successful build.

12. The system of claim 9 wherein the build service is further configured to:

    poll a source code management repository for any changes in the dependency information in component source modules; and

    in response to detecting changes in the dependency information, create the dependency graph based on the dependency information.

13. The system of claim 10 wherein the dependency graph comprises a global reverse dependency graph.

14. The system of claim 13 wherein the global reverse dependency graph is created by reversing directions of edges of a merged global dependency graph.

15. The system of claim 13 wherein the relative failure factors are calculated by determining for each of a plurality of path options in the global reverse dependency graph, a failure factor value.

16. The system of claim 15 wherein the using the relative failure factors further comprises:
if a build is optimized for failure first, then traversing the
global reverse dependency graph and performing the
build based on the plurality of path options having a
highest failure factor value; and
if the build is optimized for failure last, then traversing the
global reverse dependency graph and performing the
build based on the plurality of path options having a
lowest failure factor value.
17. The system of claim 15 wherein the failure factor value
is designated as Q, where Q is defined as:

\[ Q = P_r \times \left( \frac{\sum_{d} (P_d \times F_d)}{\sum_{d} F_d} \right) \]

where \( P_r \) represents a percentage of past failures for a
particular component source module;
\( P_d \) represents the percentage of past failure for a particular
developer (d) who committed code to the particular component source module; and
\( F_d \) represents a number of files committed by the particular
developer (d).
18. An executable software product stored on a computer-
readable medium containing program instructions for opti-

ing a build order of component source modules, the pro-
gram instructions for:

- creating a dependency graph based on dependency informa-
tion;
- using historical build information associated with previous
build failures to calculate relative failure factors for paths of the dependency graph;
- using the relative failure factors to determine an order of
   traversal of the dependency graph during a build process
   in which component binary modules are built from the
   component source modules.
19. The executable software product of claim 18 wherein
the historical build information includes at least one of:
- a history of which respective component source modules
  have failed most often as a percentage of total attempts to
  build the respective component source modules;
- a history of a developer’s contributions to build failures as
  a percentage of a total contribution of the developer;
- a total number of file changes since a last successful build.
20. The executable software product of claim 18 further
comprising:

- polling a source code management repository for any
  changes in the dependency information in the compo-
nent source modules;
- in response to detecting changes in the dependency informa-
tion, creating the dependency graph based on the
dependency information.
21. The executable software product of claim 18 wherein
creating the dependency graph comprises creating a global
reverse dependency graph.
22. The executable software product of claim 21 further
comprising creating the global reverse dependency graph by
reversing directions of edges of a merged global dependency
graph.
23. The executable software product of claim 21 further
comprising calculating the relative failure factors by deter-

ing for each of a plurality of path options in the global
reverse dependency graph, a failure factor value.
24. The executable software product of claim 23 wherein
the using the relative failure factors further comprises:
- if a build is optimized for failure first, then traversing the
global reverse dependency graph and performing the
build based on the plurality of path options having a
highest failure factor value; and
- if the build is optimized for failure last, then traversing the
global reverse dependency graph and performing the
build based on the plurality of path options having a
lowest failure factor value.
25. The executable software product of claim 23 wherein
the failure factor value is designated as Q, where Q is defined as:

\[ Q = P_r \times \left( \frac{\sum_{d} (P_d \times F_d)}{\sum_{d} F_d} \right) \]

where \( P_r \) represents a percentage of past failures for a
particular component source module;
\( P_d \) represents the percentage of past failure for a particular
developer (d) who committed code to the particular component source module; and
\( F_d \) represents a number of files committed by the particular
developer (d).