

graph. Step 714 marks the current node representing the assembly as having been handled, and the process returns to step 614 of FIG. 6 to select (step 614) and evaluate any other nodes in the graph that have not yet been handled, until none remain. Once the dependency graph is complete at step 614, the data in the dependency graph is used to construct the activation context 302, e.g., essentially by filling in the tables' fields with the correct version information for each assembly present in the graph. When constructed, the activation context is copied to the child process in the operating system data structure's defined environment process, making it the process default.

[0091] Note that to avoid problems, each configuration resolution stage is only evaluated once. For example, if a subsequent version redirect occurs as a result of any later configuration resolutions, the previous stages are not re-consulted to re-apply configuration. Re-applying configuration after other forms of configuration are applied may result in circular/infinite configuration redirects, and add unnecessary complexity to the binding process. Notwithstanding, in an alternative implementation, the process may, for example, loop back to handle a situation in which a replacement assembly may have another configuration associated therewith that can cause replacement of the currently selected assembly, and so on. Note that other safeguards against an infinite loop may be implemented to prevent a situation wherein versions have circular dependencies.

[0092] FIG. 9 shows the general steps taken to locate and load the correct version at runtime, e.g., when a program at runtime creates a global object, the system automatically gives it a version-specific named object by consulting the activation context 302 built from the manifest as altered by any configurations. Note that FIG. 9 is intentionally streamlined for efficiency, i.e., the activation context is built in advance, so that during runtime an efficient and rapid lookup can be performed to find the appropriate version.

[0093] Beginning at step 900 the activation API receives the application request including the version-independent assembly name, not the version specific name, and passes it as a parameter or the like to the runtime version matching mechanism, where it is received at step 902. If an entry for the name is in the activation context at step 904, the runtime version matching mechanism returns the version specific information (e.g., including the path and filename of the correct version) based on the manifest at step 906. If an entry for the name is not found in the activation context at step 904, at step 908 the runtime version matching mechanism returns a not found status, (or alternatively can determine and return the path and filename of the default version). At step 910, the activation API loads the appropriate version, and returns a loading status or the like. The operating system also maps any uses of this named object to the appropriate version to allow for multiple versions of the code module to run simultaneously without interfering with each other, whereby, for example COM object data is isolated per object. At this time, the correct version as specified in the manifests is loaded, even though the application's executable code did not specify any version. Indeed, by providing an associated manifest that can be stored into the application's directory, an already existing application (e.g., written and installed before the present invention) can benefit from the present invention. In this manner, the application runs with a controlled set of assemblies bound thereto.

[0094] Turning to the second alternative mode, FIGS. 10-11 represent example steps that may be taken, such as dynamically during runtime, to determine which version of an assembly to bind. To this end, the binding mechanism 305 of FIG. 3B (e.g., of the operating system) may check for an application manifest in same file system directory as the calling executable, as represented in FIG. 10 by step 1000. If an application manifest does not exist, the binding mechanism 305 handles its absence in another manner, (step 1002), e.g., the operating system essentially will give the application default versions during runtime, such as by first loading any requested component or assembly from the application's own directory when one is present, and otherwise using the default assemblies from the assembly cache.

[0095] When step 1000 determines that an application manifest exists, it is interpreted as represented by step 1004. Then, in this second mode, step 1006 tests whether an application configuration exists for this application, e.g., in the application directory. If not, step 1006 branches ahead to FIG. 11 to test for whether a publisher configuration applies, as described below. If an application configuration is found at step 1006, step 1006 branches to step 1008 wherein the application configuration is interpreted to determine whether there is an instruction therein for replacing the assembly version that is currently under evaluation. If such a relevant replacement instruction is found, step 1008 branches to step 1010 wherein the current assembly binding information is replaced.

[0096] FIG. 11 represents the next steps in this second alternative mode, wherein if an application configuration exists, step 1100 tests whether it includes data specifying the safe mode of configuration resolution. If the safe mode is specified, step 1100 avoids the publisher configuration evaluation by branching ahead to test for administrator configuration, described below with respect to steps 1108, 1110 and 1112. Note that although not specifically shown in FIGS. 10 and 11, step 1006 (FIG. 10) can branch to step 1102 (FIG. 11) when no application policy exists, unless something other than the application configuration (e.g., an administrative setting) is capable of setting the safe mode.

[0097] If the safe mode is not specified at step 1100, step 1100 branches to step 1102 to test for a publisher configuration. If a publisher configuration is found at step 1102, step 1102 branches to step 1104 wherein the publisher configuration is interpreted to determine whether there is an instruction therein for replacing the assembly version that is currently under evaluation, either as originally specified in the manifest or as replaced by application configuration (as described above with respect to step 1010). If a replacement instruction is found at step 1104, step 1104 branches to step 1106 wherein the replacement is made, otherwise step 1104 effectively bypasses step 1106.

[0098] The process continues to step 1108, which represents the start of the second mode's third phase of the configuration resolution process, wherein a test is performed to determine whether the system has an administrator configuration, e.g., in the system directory. If not, step 1114 is executed, as described below. If so, step 1108 branches to step 1110 wherein the administrator configuration is interpreted to determine whether there is an instruction therein for replacing the current assembly version with another version. If a replacement instruction is found, step 1110