

[0081] compiler or optimization tool to have a precise description of the code in the artifact and the environment in which it executes;

[0082] error-detect tool to have a precise description of the code in the artifact and the environment in which it executes;

[0083] The manifest need not contain all available metadata about a program or system, but it needs to provide sufficient information to enable reliably locating additional metadata. In one embodiment, for example, the binary load modules (EXE, DLLS, etc.) for a program contain metadata that references metadata associated with specific load modules. In this embodiment, the manifest informs the system of the existence of this additional metadata within the load modules.

[0084] In one embodiment, the manifest identifies the type of each subcomponent of the manifest. The subcomponent type identifies a piece of helper software, which knows how to interpret the contents of the subcomponent, extract additional metadata from the component, and derive additional metadata about the component.

[0085] For example, in one embodiment, load modules described in a manifest are expressed in an abstract instruction set that allows a verification tool to determine if they obey certain software properties, such as conformance to communication requirements. The manifest for the load modules identifies the exact abstract instruction set used for each load module so that the system verifier 164 can determine which helper software to load to verify specific system properties, such as the communication requirements.

[0086] In yet another aspect of an embodiment, the information used to determine compatibility among the parts of a system is delivered independently of the components, as well as along with them. This information can arrive from many sources, and a local administrator or agent can define or follow rules for disambiguating partial or contradictory information.

[0087] In yet another aspect of an embodiment, the information used to determine compatibility among the parts of a system changes over time, as new information becomes available at an appropriate location, or as old information is revoked.

[0088] Manifests may be combined into graphs to describe arbitrarily complex software systems. Manifests may refer to external manifests as dependencies. Manifests may also contain embedded manifests. In one embodiment, the manifest for an application contains or uses manifests for sub-components of the application.

[0089] Depending on packaging decisions made by the publisher of the application, subcomponents can be either embedded in the manifest or referenced as external entities. In one embodiment, external dependencies include a name and version number of the dependency, or other clarifying information. In another embodiment, external dependencies are named through a signed digest. In another embodiment, this information can be updated, revoked, and clarified (i.e., disambiguated).

[0090] In one embodiment, the manifest for an application is packaged and delivered with its associated application. In another embodiment, the manifest for an application is

packaged and delivered separately from its associated application. With this, the presumably multiple components of an application may be delivered separately and after delivery of its associated manifest.

[0091] In one embodiment, external manifests may also be referenced as sources of external information. These external manifests may be named individually, or they may be named as members of a group.

[0092] There are two forms of the manifests: static and dynamic. The static manifests are stored in association with software artifacts. The dynamic manifests are employed during the runtime of an executable component associated therewith. The dynamic manifest includes the static metadata (which is still available at runtime) and additional dynamic metadata that are constructed at runtime to connect runtime system elements, like processes and operating system objects.

[0093] This aspect further enables bridging from low-level implementation concepts to higher-level concepts. The self-describing feature of the software artifacts is useful on a running and active system and not just a static system. For example, the “well-formedness” and or consistency of a running system of processes can be verified similarly to the verification of the system image.

[0094] In one embodiment, “über” manifests describe all software available on the computer 102, directly or indirectly, and whether such software is installed or not.

Methodological Implementation of Exemplary Self-Describing Artifact Management and Gatekeeping

[0095] FIG. 2 shows a method 200 performed by the self-describing artifact manager 160 and/or the execution gatekeeper 162. This methodological implementation may be performed in software, hardware, or a combination thereof. For ease of understanding, the method is delineated as separate steps steps should not be construed as necessarily order dependent in their performance. Additionally, for discussion purposes, the method 200 is described with reference to FIG. 1.

[0096] At 210 of FIG. 2, the self-describing artifact manager 160 facilitates persistence of manifests with their associated artifacts. As shown in of FIG. 1, manifest 132 is persisted in association with the systems artifact 130 or it is stored at some derivable or known-location on the storage device 120. Similarly, manifests 142 and 152 are stored in association with application artifacts 140 and 150.

[0097] At 212, the self-describing artifact manager 160 updates the self-describing artifacts of a system in accordance with the changes in the system’s content and/or configuration. Such changes may be result of, for example, installation of new content, a manual configuration change, and automatic configuration change performed by the operating system. Before they are applied, updates may be checked in the context of the collection of system manifests to ensure that if applied they will result in a viable system.

[0098] At 214, the self-describing artifact manager 160 optimizes the use of artifacts for execution. The manager can determine which load modules will be combined in processes for an application. The manager can then combine load modules into a smaller number of load modules, which have been optimized together. Similarly, using the system