

[0122] A property is compositionally verifiable if components can be verified for the property individually, and when composed, the system can be known to maintain the same property without re-verifying the all of the components. For example, in programming systems type safety is considered compositionally verifiable if individual load modules can be verified as type safe and when legally combined they maintain type safety. In this case, the system inspector **180** can verify that each load is type safe and then verify that the load modules are combined in a legal manner without requiring complex verification across the entire system whenever a new load module is added.

Methodological Implementation of Exemplary System Inspection

[0123] FIG. 4 shows a method **400** performed by the system inspector **180**. This methodological implementation may be performed in software, hardware, or a combination thereof. For ease of understanding, the method is delineated as separate steps represented as independent blocks in FIG. 4; however, these separately delineated steps should not be construed as necessarily order dependent in their performance. Additionally, for discussion purposes, the method **400** is described with reference to FIG. 1.

[0124] At **410** of FIG. 4, the system inspector **180** obtains a copy of an offline “system image” of a software-based computer, such as the computer **102**. This action is illustrated in FIG. 1 by the large-headed arrow.

[0125] At **412**, the system inspector **180** performs an analysis of the offline system image to verify conclusively that the computer **102** contains specific functional components (such as the OS or applications). More particularly, the inspector examines the self-describing artifacts to see if all of the necessary components (described as such and referenced by manifests of the self-describing artifacts) are located and properly identified.

[0126] At **414**, the inspector reports the results of this analysis.

Abstractions

[0127] Operating systems provide abstractions to frame computation and allow programmers to create software more easily by focusing more completely on their domain of expertise. An abstraction denotes a model of one or more components that represents the essential characteristics of those components that distinguish them from all other kinds of components and thus provides crisply defined conceptual boundaries.

[0128] Examples of existing operating system abstractions include file system abstractions to control and manage storage, I/O abstractions to control I/O devices, Graphical User Interface (GUI) abstractions, process abstractions to hold computation, and interprocess communication (IPC) abstractions to enable communication between processes.

[0129] Without these basic abstractions, programmers would be forced to devise their own ad hoc methods for performing common tasks. Invariably such diverse ad hoc methods lead to reduced programmer productivity, large scale duplication of effort, and increased system errors.

[0130] In, at least, one implementation, the exemplary self-describing artifact architecture creates new operating system abstractions, which include:

[0131] a system prototype, which represents a “runnable” (e.g., executable on the computer **102**) software system including operating system and programs;

[0132] a system abstraction, which represents an active or “running” system including the operating system and programs;

[0133] an application prototype, which represents a runnable application program;

[0134] an application abstraction, which represents an instance of an active or “running” program; and

[0135] a process prototype, which represents a runnable process.

[0136] FIG. 5 shows an example structure **500** of a software-based computer capable of implementing this new architecture. This includes the new abstractions and prototypes introduced by this new architecture in the context of conventional abstractions and prototypes. From top to bottom of FIG. 5, the four tiers in this example structure **500** are:

[0137] 1. Systems tier **510**, which includes:

[0138] an system abstraction **512** (which includes the operating system components (e.g., scheduler, IPC manager, I/O manager, security manager, garbage-collection and memory manager, etc.), and any other systems level components) and by inclusion all applications;

[0139] systems prototypes **514** associated with at least one system;

[0140] systems manifests **516** associated with at least one system including the OS and applications.

[0141] 2. Application tier **520**, which includes:

[0142] One or more application abstractions **522**;

[0143] application prototypes **524** associated with at least one of the application abstractions;

[0144] application manifests **526** associated with at least one application-based artifact.

[0145] 3. Process tier **530**, which includes:

[0146] One or more process abstractions **532**;

[0147] Process prototypes **534** associated with at least one of the process abstractions;

[0148] process manifests **536** associated with at least one process-based artifact.

[0149] 4. Load source tier **540**, which includes load source manifests **546** associated with at least one load-source artifact (e.g., load module).

[0150] For completeness, the system model may also include a system abstraction representing the entire running software system. In practice, the operating system **512** itself typically acts as the system abstraction.

[0151] Manifests are declarative description a software component (e.g., process, application, or OS element). When the manifests and the software components are persisted in association with each other, then the components are self-describing artifact.