Singularity Design Note 12

The Application Manifest

Install, update, and uninstall as OS features.

Current operating systems contain abstractions for many components of modern applications, but the application itself remains unrepresented. This situation leads to inefficiencies in running and processing applications and their data. In particular, with knowledge of the application as a first-class citizen in the system we could provide significant stability improvements in updates, installs, and uninstalls. This document describes an important component of the application abstraction, the application manifest, in the context of Singularity and motivates its uses.

1. Introduction

An application can be represented as a simple combination of declarative statements referring to resources and their dependencies. Unfortunately, the current model of the application in modern operating systems is simply the process. Objects of coarser granularity are only implicitly represented by metadata and files scattered throughout the system. This lack of a coherent representation has led to significant difficulty and expense at Microsoft and elsewhere in deploying complex applications. It is well known that adding or removing an application can cause supposedly unrelated software to break. Several components of Singularity would benefit from knowledge of the application as a first-class citizen in the OS. The two most obvious examples are setup and update. Since the Singularity model prefers that our boxes require zero-administration, we must be able to send out updates with confidence that they will not change the state of unrelated applications. Similarly, the security model under Singularity requires knowledge of application boundaries, dependencies, and identities to enforce access policies and rights. These properties can be described in terms of the components of an application, which are simply the resources in the system along with the dependencies between them.

Although many current setup languages mix declarative and imperative components, we claim that all installation actions can be represented as declarative statements. We choose to represent in our application manifest only the final state of the application after install or update, and leave the process of constructing this state in the system to the Singularity installer. We ensure that enough information exists in our manifest for the installer to deduce the correct installation steps and to decide whether or not the installation has succeeded.

1.1. Goals

The principal goal of the application manifest is to allow the operating system to add, update, and remove applications without changing the correctness of any other application. A secondary goal is to make application dependencies more explicit. We want the dependencies to contain more information than strong, versioned names, even though we recognize that we can never capture the full semantic dependencies between two applications. It is in general impossible to specify the complete dependencies between two applications: these dependencies can be tied up in the details of the implementation (and even bugs) of each. We rely instead on the typed structure of Singularity to give us stronger guarantees about the correctness of an update or

installation. The application manifest packages this type information so the operating system can infer the effects of adding, modifying and removing files.

1.2. Preliminaries

Before we can state the exact structure and mechanisms related to the application manifest, we first describe several fundamental assumptions and definitions used in this document

1.2.1. Definitions

- **Internal Type:** An MSIL type
- **External Type:** An Exchangeable Type
- **Component:** An instance of an external or internal type.
- **Module:** A component containing code. While a module is typically an MSIL assembly, multiple modules may be packed into a single assembly.
- **Module Interface:** The set of MSIL types on a module (assembly) marked as public. The module interfaces includes not only public classes, structs, enums, and interfaces, but also public methods, properties, fields, and constants. The module interface may include additional information such as Spec# pre and post conditions and invariants.
- **Channel Contract:** A machine readable specification of the contract governing the types of message and allowed exchange patterns on a class of Singularity channels.
- **Statement:** A channel contract or module interface.
- **Resource:** Currently a synonym for component. Although there are resources (e.g. hardware devices) that are not typed collections of bits on the system, we still describe them with an application manifest generated by a device driver or a plug-and-play system.
- **Public Component:** A component visible to some component outside the current application.
- **Private Component:** A component visible to no components outside the current application.
- **Application Surface:** The set of all channel contracts either imported or exported by the application.
- **Metadata:** Data describing the structure of code or other data; any public, externally-typed data. Metadata is public because it is meant to be visible to more than one application. Metadata is externally typed because the data is to be read by other applications' processes, and thus can only be read through a channel.
- **Namespace:** A function that maps a string into a resource.

1.2.2. Namespaces

Since a namespace maps a string into a resource, the same string may produce different resources in two different namespaces. For example, the string "Word" in the application configuration name space maps to the settings for the Word application, whereas the same string in the file system maps to a directory contain the modules for the Word application. We assume that in Singularity there is a single, global namespace that aggregates all other namespaces into a hierarchy. For example, the filesystem is a subnamespace of this global namespace, as is the Application namespace used to find programs. Each application is created with its own protected namespace, over which it has complete control.

2. The Manifest

Every application is composed entirely of resources and dependencies between these resources. The **application manifest** describes the resources contained in the application and their dependencies. In this section we describe the components of a manifest in Singularity.

2.1. Resources

In Singularity every resource is contained in a namespace, and further that every resource is typed, at least in some internal type system. There are three fundamental classes of resources: **code**, **data**, and **namespaces**. Although it is clear that there is no distinction between code and data in the abstract, most apps will only treat a given set of bits as one of code or data (for instance, Notepad treats a JScript file as data, whereas WScript treats it as code. Furthermore, the conversion from data to code should be carefully controlled. If an application is ever going to treat a resource as code, we require that it specify in the manifest that the resource is a code resource. Before installation, all code resources are subject to system verification to prove their validity.

For all resource statements in the application manifest, we assume the existence of a security policy language that can be used to specify policies such as trust relationships for each resource in the manifest. The exact form of this policy language remains to be decided by the security team.

2.1.1. Code

The application manifest divides code into two groups of modules: modules that form the root of a new process, and modules that do not. We refer to the former as a **process module** and the latter as a **library module**. An application manifest must contain at least one process component, but can contain an unbounded number of process modules, one for each process created within the application. Historically, process modules and library modules are called EXEs and DLLs respectively, and have differing invocation interfaces. In Singularity, process modules and library modules have the same invocation interface, only their role in the application manifest that makes them distinct.

Each part of the application manifest that refers to a process or library module makes statements about the external surface and internal structure of that module. The external surface includes the module interface exposed within the process and the channels exposed outside the process. Statements about the internal structure of the module typical name the module with specificity, such as fully qualified name, a version, and a secure hash of the module. The naming may include a level of indirection so that a publisher of a manifest may provide a subsequent release incorporating a bug fix. Naming may also be modulated by system policy.

For example, a process module named floppy.sys that is used to construct the process in Singularity that runs the floppy disk driver will have an external interface of channel contracts and endpoints. Similarly, a library module will have an external interface of MSIL type information with Spec# annotations. The Spec# annotations and channel contracts allow system to check for not just naming conformance, but also semantic conformance, when verifying compatibility and resolving dependencies.

2.1.2. Data

Data is any component that cannot be interpreted as a module. For example, metadata is public, externally-typed data, as discussed above, and an element of metadata will show up in an implementation of the application manifest as a named value with a type specification (along with a namespace that contains it, as will be discussed below).

The application manifest also supports the concept of a data **collection**. A collection may be ordered or unordered. It may also specify that the members of the collection must satisfy some statement. Ordered collections further support the concepts of a first and last element, as well as relative ordering between two elements. Its elements can either be constrained to a fixed set specified at application install time or not.

2.1.3. Namespaces

In the application manifest, names are mappings either to an instance of a type or to another namespace (which can be thought of as an instance of the type **namespace**). As noted in section 1.2.2, there is a global namespace in Singularity, and all other namespaces are contained in that namespace. We assume the existence in the global namespace of certain well-known namespaces. One is the Filesystem, under which a filesystem structure can be stored. Another top-level namespace is the Application namespace, under which each application's namespace is constructed upon installation. This namespace will contain the private and public names for the application.

2.2. Dependencies

We represent all dependencies by the *requires* relation. This relation expresses a typed dependency of one component on another. In the application manifest, this relation is typed with a contract (for resources, like other processes or metadata, accessed through external types) or Spec# statements (for resources, like libraries, accessed through internal types).

The *supplies* relation is the inverse of the requires relation; each resource specified in an application manifest provides a channel contract (for resources accessed from another process) or a set of module interface (for resources accessed within a process). The set of externally-typed supplies dependencies forms the surface of the application.

Requirements may be optional. An example of an optional requirement is the existence of a printer in Word. Word would prefer to have a printer, but documents can be can be created and edited without one. Since these optional requirements have no bearing on the correctness of the application, their presence in the manifest is only to help the operating system; a correct application will only access resources that it has listed in the manifest. The operating system will enforce manifest declared resource access.

2.2.1. Uses of Relations

The main use for the requires and supplies relations is to check that the correct resources are available for a given application when it runs. The current troubles in the application space are caused by not having enough semantic information to determine whether or not to install a given resource. These questions cannot simply be solved by version numbers and side-by-side installations of different files, because updates may, for example, change the statements associated with a given resource. During installation, update, and uninstall we must check that the application is still consistent and that all other applications that depend on this particular resource still have their dependencies satisfied.

3. Uses of the Application Manifest

We have developed a XML-based language, called **dasl** (the Declarative Application Specification Language) (pronounced “dazzle”) that allows us to give a declarative definition of an application. In this section, we present this language and the conversion of a simple example taken from the componentization in the lab01 NT source tree.

3.1. Dasl elements

- `<public>`: This element contains all the public components of the application.
- `<private>`: This element contains all the private components of the application.
- `<code>`: This element names a module in a namespace. It also contains any number of `<requires>` elements that specify the requirements of this module. This element must contain a `namespace` attribute to specify the namespace in which it lives, and it must contain a `name` element that names this module uniquely in this namespace.
    - The attribute `main` means that this is a process component and the root of the process tree for this application. When the application is started in Singularity, this process will be created and allowed to create other processes as needed.
    - The attribute `process` means that this is a process component.
    - The attribute `sealed` on a main or process component
- `<data>`: This element represent a data component in a namespace. This element must contain a `namespace` attribute to specify the namespace in which it lives, and it must contain a name attribute to uniquely name this component in its namespace. If it is contained in a `<collection>`, then it may have `<requires>` elements that state its positioning requirements in that collection.
- `<namespace>`: This element represents a namespace and is either contained in another `<namespace>` element or has an explicit `parent` attribute that names its parent namespace.

- `<requires>`: This element represents a requirement as described in section 2.2.
  - The attribute `optional` may be `true` or `false`. The default is `false`.
  - If this element is contained in a `<data>` element in an `ordered` `<collection>`, then it may have the attribute `before` or `after`. These attributes say that this required item must come `before` or `after`, respectively, the containing `<data>` element in this collection.
  - If this element is contained in a `<data>` element in an ordered `<collection>`, then it may have the attribute `first` or `last`. These attributes say that this required item must come `first` or `last`, respectively, in the collection. The `first` attribute is analogous to a `before` all. The last attribute is analogous to an `after` all.
- `<collection>`: A collection may take the place of any data item. Each collection has <data> elements as children. It must contain a `namespace` attribute that says where in the namespace this component lives. It also has a `name` attribute that specifies the name of this collection. Since the application manifest is entirely declarative there are only a few possible attributes.
  - The attribute `complete` may be either `true` or `false`. If it is `true`, then the list of `<data>` elements is the complete list of possible elements in this collection. It will both replace any (optional) collection or data element in its place in the namespace, and will not allow other applications to add new entries to the list. If it is `false`, then the `<data>` elements will be added to any collection that might already be using this place in the namespace (if possible), and other elements may be added by other applications, subject to any constraints imposed by the `<requires>` tags in each `<data>` element, and subject to the `type` attribute below, if there is one.
  - The attribute `type` contains a statement that specifies what type of data components are allowed in this collection.
  - The attribute `ordered` states whether or not this collection is ordered.

3.2. Install

3.2.1. Examples

The simplest example of all is notepad, which has only one file, and a very simple set of dependencies and namespaces. We have, however, compared our set of resource specifications with that of the Windows Installer XML (WiX) toolkit. This toolkit is currently used inside Microsoft and was published recently on SourceForge (http://wix.sourceforge.net). It allows the specification of application installation information in an XML format that is then compiled to an MSI. We have found that every tag in WiX that corresponds to a resource has a reasonable correlate inside dasl. We have also found that all the actions in WiX can be deduced from the dasl specification along with the types of the resources and their dependencies as specified in our XML format. Since dependencies are currently mostly implicit in Windows, we do not have enough information to write a complete dasl specification of Notepad, but the following shows a full conversion of the CMI manifest.

3.2.1.1. Notepad

```
<?xml version="1.0" ?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="dasl.xsd"
             identity="Notepad">
    <public>
        <data namespace="Notepad" name="DefaultFont"
              type="external:string" />
    </public>
    <private>
        <code namespace="Notepad" main="true"
              name="notepad.exe" type="version=1.0">
            <requires namespace="Applications" name="Windows Shell"
```



```
                    type="application"/>
        </code>
        <data namespace="Notepad/Settings/Font"
              name="Name" type="external:string" value="Arial" />
        <data namespace="Notepad/Settings/Font"
              name="Size" type="external:int" value="10" />
        <namespace name="Settings" parent="Notepad">
            <namespace name="Font"/>
        </namespace>
    </private>
</application>
```

3.3. Update

An update to an application simply takes the form of additions and removals of resources and dependencies. The installer then needs to check that the new state of the namespace tree still allows all applications present to function correctly according to their type specifications. We support update via dasl, and in this subsection we provide an example of what an update to notepad might look like.

3.3.1. Dasl again

In a dasl update, the most important result is that the final state of the application after the update should correspond exactly to what is given in the update manifest. In the spirit of a declarative application manifest language, we do not provide operations to explicitly add or remove elements. Instead, an update consists of a new specification of what the state of the application should look like after the update has occurred. We allow the installer to deduce the appropriate changes to get the application into the correct state.

3.3.2. Examples

We can imagine the following (admittedly contrived) update to notepad where a new surface dll was added and the namespace was made public.

3.3.2.1. Notepad

```
<?xml version="1.0" ?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="dasl.xsd"
             identity="Notepad">
    <public>
        <data namespace="Notepad" name="DefaultFont"
              type="external:string" />
        <code namespace="Notepad" name="notepadCtl.dll"
              type="notepadContract"/>
            <requires namespace="Kernel" name="component.dll"
                      type="aContract"/>
        </code>
        <namespace name="Settings" parent="Notepad">
            <namespace name="Font"/>
        </namespace>
    </public>
    <private>
        <code namespace="Notepad" main="true" name="notepad.exe"
              type="version=1.0">
            <requires namespace="Applications" name=" Windows Shell"
                      type="application"/>
        </code>
        <data namespace="Notepad/Settings/Font" name="Name"
              type="external:string" value="Arial" />
        <data namespace="Notepad/Settings/Font" name="Size"
              type="external:int" value="10" />
    </private>
</application>
```


3.4. Uninstall

To uninstall an application using dasl, we provide an application manifest that matches exactly an installed application. Once dasl confirms that this application is installed, it uses the declarative statement of what exists in the application to remove all of its elements. If any applications are installed that are using the same namespaces and name/value pairs as this application, these namespaces and name/value pairs are not removed. Since we know that the state of the namespaces after install is consistent with all installed applications, it is safe to leave the namespaces and name/value pairs as they are; in fact, we know that they are still required.

3.5. Invocation

Given the above application manifest, we can instantiate applications in Singularity that match a particular installed manifest. The installed manifests all live in some manifest namespace. When a user requests that an application be run, the appropriate manifest is chosen by application identity from the set of manifests. This manifest contains the information about the process that will be started by the runtime, since we require that there be exactly one `<code>` element with a `main` attribute. The system then instantiates the main process using the code object with the `main` attribute, which runs as the first node in the process tree. The system can dynamically ensure that the application follows exactly the resource dependencies in its manifest, by checking that it only uses resources it claims to require.

It is clear that, as it is started, each process can be marked with the application instance identifier with which it is associated. Then the system can determine the application with which a given process is associated simply by examining the process itself.

4. The Intermediate Application Manifest

4.1. Motivation and Goals

In order to facilitate the rapid identification of manifest requirements, Singularity currently uses a less general manifest format. This format satisfies the following goals:

4.1.1. Simple Subtree Extraction with Guaranteed Structure

The intermediate format allows for tags with specific names, such as “driverCategory,” “ioPortRange,” and “imp”. Furthermore, the current manifest generation tool, mkmani, guarantees a particular structure to these tags. As a result, the Kernel can use a simple Xml reader to extract tags without extensive error-handling (for example, it can assume that there will be exactly one “propertySet” tag).

4.1.2. Support for Install-Time Configuration

Ultimately, the content of an application manifest will change upon installation. Certain values may be set, such as default parameters, app locations, and endpoint names. Additionally, the installed manifest will store information explaining what MSIL assemblies were shipped as an application, how they were verified, and what cached native-code file ought to be run in place of the MSIL assemblies.

As Singularity currently lacks an installer, the structure of this information is not well-defined, and little of this information is even known. Consequently, we require a flexible format for representing the existing pieces of this information in a manner that is accessible in a Singularity system that lacks an installer.

4.1.3. Dasl Compatibility

Despite the support for programmer-defined names for tags and attributes, the intermediate application manifest is ultimately compatible with dasl. We expect that upon completion of a dasl parser and schema validator for Singularity, the conversion from existing manifests to dasl manifests will be straightforward and simple.

4.2. Structure

The intermediate application manifest (IAM) format consists of 6 blocks of information. Their format, semantics, and use are detailed below:

4.2.1. The <manifest> Tree

To distinguish from the dasl format, the root tag of the IAM format is named “manifest”. This tag is the parent of the remaining trees described in this section.

```
<?xml version="1.0" encoding="us-ascii"?>
<manifest>
 <application> ... </application>
 <assemblies> ... </assemblies>
 <signature> ... </signature>
 <propertySet> ... </propertySet>
 <category> ... </category>
 <driverCategory> ... </driverCategory>
</manifest>
```

4.2.2. The <application> Tree

The <application> tree mirrors the dasl <application> tag:

```
  <application identity="DiskDrive">
    <properties>
      <code main="True" path="DiskDrive.x86" />
    </properties>
  </application>
```

The application “identity” property provides a public name for the application. This field currently names the subtree of the /init namespace where the application and its manifest reside.

The code “main” property is not used at this time, and is set to “True” in all manifests.

The code “path” property names the .x86 file that Singularity is to run. This property is currently generated at compile time during the creation of a manifest. In order to use this field, it is then updated at run time to reflect the full name of the .x86 file, currently /init/identity/path. This property should not ultimately be set by the compiler. Instead, it should be set during application installation, as the native code file will not be created until this time.

4.2.3. The <assemblies> Tree

Anticipating the day in which Singularity verifies and installs applications, the IAM includes a list of all the assemblies used by the application. Bartok will use these files to create the native-code .x86 file that Singularity executes.

```
  <assemblies>
    <assembly filename="DiskDrive.exe" />
    <assembly filename="Namespace.Contracts.dll" />
    <assembly filename="Io.Contracts.dll" />
    <assembly filename="Corlibsg.dll" />
    <assembly filename="Singularity.V1.ill" />
    <assembly filename="Corlib.dll" />
    <assembly filename="System.Compiler.Runtime.dll" />
    <assembly filename="Microsoft.SingSharp.Runtime.dll" />
    <assembly filename="ILHelpers.dll" />
  </assemblies>
```

These entries are not complete, lacking the full assembly information for each entry.

4.2.4. The <signature> Tree

The signature tree is currently empty.

```
  <signature />
```


4.2.5. The <propertySet> Tree

The propertySet tree is a rough approximation of the configuration that is constant for all executions of an application. Currently, this tree holds five types of entries: endpoints, strings, integers, lists of strings, and list of integers:

```
<propertySet>
  <endpoint> ... </endpoint>
  <extension> ... </extension>
  <serviceProvider> ... </serviceProvider>
  <configString name="foo" optional="False" default="bar" />
  <configInt name="MaxThreads" optional="False" value="16" />
  <configStringList name="hosts" optional="False" />
  <configIntList name="ports" optional="False" />
</propertySet>
```

For each of the four tags prefixed with “config,” the role of the tag is to identify a value or set of values. The properties of the tag specify a name for each field (name), whether a configuration field may be skipped or not (optional), and an optional default value of the property (value). Default values for a list will be represented by adding children to the configStringList or configIntList tag.

Singularity currently does not use these elements of the propertySet. They exist strictly to initiate analysis of what configuration parameters an application needs, and how these parameters can be declared and provided to an application.

Discussion of the three endpoint tags is deferred until section 4.2.7.

4.2.6. The <category> Tree

A manifest may hold multiple category trees. Each tree receives a name, so that an application can be invoked in one of many roles, with the configuration for each role being declared statically.

```
<category name="DefaultMode">
  <configString name="Title" optional="True" value="Play" />
</category>
```

A category tag can hold any child that is legal for a propertySet. Ideally, every application should have at least one category, with one category specified as the default. This is not currently implemented in the metadata, as categories are not currently used by Singularity. The “name” of a category enables the application invoker to specify uniquely which category he wishes to instantiate.

4.2.7. The <driverCategory> Tree

Strictly speaking, the driverCategory is a descendent of the category. It identifies that an application can be invoked as a device driver, and specifies the exact and complete set of resources through which the application will interact with the outside world. An application can declare multiple driverCategories.

```
<driverCategory>
  <device signature="/ide/controller" />
  <ioIrqRange baseAddress="14" rangeLength="1" Shared="True" />
  <ioDmaRange baseAddress="496" rangeLength="2" Shared="True" />
  <ioMemoryRange baseAddress="1012" rangeLength="2" Readable="False" />
  <ioPortRange baseAddress="65440" rangeLength="2" Writable="True" />
  <region />
  <extension> ... </extension>
  <serviceProvider> ... </serviceProvider>
  <endpoint> ... </endpoint>
</driverCategory>
```

The entries in the driverCategory tree fall into three broad categories:

4.2.7.1. Device Description

If an application is a device driver for a hardware device, it must declare all of the device signatures it is capable of serving. This is accomplished through the device tag. The required property of this tag, "signature," identifies the prefix of the signature of a device that the driver can serve.

In addition, a device signature may include the tag "CreateIfAbsent=True". This tag is specifically intended to address a quirk in the way a Pnp bus enumerates devices. Under limited conditions (which currently appear only for Singularity on the VirtualPC), objects for a Pci bus's resources won't be created during enumeration. This tag indicates that for this device, such behavior is acceptable and the resource manager should manually create those resources using the default values from the metadata.

4.2.7.2. Io Resource Requirement Declarations

An instance of a device driver serves a single device whose signature matches the prefix specified by the device description tag. Singularity also performs a simple sanity check by ensuring that device metadata matches the set of dynamic hardware resources identified by device enumeration. In addition, the driver may specify other fixed hardware resources it needs in order to operate properly. Further details are available in SDN 24.

4.2.7.2.1. Tags

There are four basic Io resource tags, ioIrqRange, ioDmaRange, ioMemoryRange, and ioPortRange. These four tags correspond to a device driver using ranges of irqs, dma channels, memory, and ports. For each of these tags, there are two required properties, baseAddress and rangeLength. In addition, there are two optional parameters, Fixed and Shared. Both of these default to false unless they are specified as being "True".

The ioMemoryRange and ioPortRange tags also support properties to limit their accessibility. The tags AllowRead and AllowWrite default to "True" unless they are specified as being false.

There is a fifth Io resource tag, used to identify special memory resources for a device driver. This tag, "region," has three attributes to specify restrictions on the types of memory regions to be allocated. The attributes are addressing, alignment, and size.

4.2.7.3. Endpoint Requirement Declarations

There are three endpoint tags, "extension", "serviceProvider", and "endpoint." These tags differ only in name, and so in the interests of brevity only "endpoint" is discussed here. All three tags are valid children of propertySet, category, and driverCategory tags:

```
<endpoint startStateId="3"
          contractName="Microsoft.Singularity.Io.VolumeManagerContract"
          endpointEnd="Imp"
          assembly="Io.Contracts"
          version="0.0.0.0"
          culture="neutral"
          publicKeyToken="null">
 <imp>
  <inherit name="Microsoft.Singularity.Channels.Endpoint" />
  <inherit name="Microsoft.Singularity.Io.VolumeManagerContract.Imp" />
 </imp>
 <exp>
  <inherit name="Microsoft.Singularity.Channels.Endpoint" />
  <inherit name="Microsoft.Singularity.Naming.ServiceContract.Exp" />
  <inherit name="Microsoft.Singularity.Io.VolumeManagerContract.Exp" />
 </exp>
</endpoint>
```

These tags identify to Singularity that the kernel is to bind an endpoint according to system policy prior to starting an application. In order to do so, the attribute "startStateId" identifies the integer value of the start state of the contract, the property "contractName" gives the full name of the contract type, and the "imp" and "exp" trees enumerate the full inheritance hierarchy of each endpoint within the contract. The "endpointEnd" tag identifies which endpoint is to be given to the application.

The remaining fields are properties of the assembly in which the contract is declared. These fields are not currently used.

