RFC 154 - Generic Capabilities & Requirements

Final

17 Pages

Abstract

This document proposes an approach for bundles to declare generic capabilities and for other bundles specify requirements on those capabilities.

0 Document Information

0.1 Table of Contents

0 Document Information.......... 2
0.1 Table of Contents.......... 2
0.2 Terminology and Document Conventions.......... 3
0.3 Revision History.......... 3

1 Introduction.......... 3

2 Application Domain.......... 4

3 Problem Description.......... 4
3.1 Extender bundle dependencies.......... 4
3.2 Operating environment dependencies (Bug 282).......... 4
3.3 Limitations of Bundle-RequiredExecutionEnvironment.......... 5
3.3.1 Inability to capture java.* dependencies (Bug 390).......... 5
3.3.2 Lack of expressiveness (Bug 1000).......... 5
3.4 Native code dependencies (Bug 1013).......... 5

4 Requirements.......... 6

5 Technical Solution.......... 6
5.1 General approach.......... 6
5.2 Optionality.......... 7
5.3 Effective time.......... 7
5.4 Management Agents.......... 7
5.5 Resolution result.......... 8
5.6 Multiple matching providers.......... 8
5.7 Cardinality of requirements.......... 8
5.8 Capability/Requirement verification.......... 9
5.9 Host/fragment conflicts.......... 9

6 Considered Alternatives.......... 9
6.1 Discovery.......... 9
6.2 Uses constraints.......... 9
6.3 Singleton capabilities.......... 9
6.4 Mappings for existing headers.......... 9
6.5 Bundle state related capabilities.......... 10

7 Security Considerations.. 10

8 Document Support.. 10

8.1 References.. 10
8.2 Author's Address.. 11
8.3 Acronyms and Abbreviations.. 11
8.4 End of Document.. 11

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 8.1.

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	Dec. 14th, 2009	
Southampton F2F feedback	Mar. 1st, 2010	Adopted typed attribute approach. Resolve result is now observable with lifecycle coupling. Added wording on multiple matching providers, capability/requirement verification, and existing header mappings.
Bundle state feedback	Mar 16th 2010	Added framework property that controls which capability namespaces get processed by the framework and commentary on bundle state related capabilities
Mountview F2F feedback	May 13th, 2010	Rolled back the prior changes, added language to indicate the resolve-time scope of the solution, and added initial requirements.
London F2F feedback	June 1st, 2010	Add "effective time" directive and adjusted scope discussion accordingly
RFC 151	July 23rd, 2010	Added a simple comment to tie this RFC to RFC 151 for purposes of an API for reflecting on wiring.
Ottawa F2F feedback	August 25th, 2010	Added "uses" directive, added List attribute type, and more namespace clarifications.
CPEG call feedback	August 30th, 2010	Added note about case sensitivity, defined List syntax, clarified some host/fragment issues, added permission.
Final Draft	09/02/10	Updated the grammar for the manifest headers to be more in line with existing specified manifest headers.

1 Introduction

The OSGi specification defines a handful of different dependency types that can exist among bundles, such as package, bundle, and fragment dependencies. All of these dependencies relate to defining code visibility for the involved bundles. The OSGi framework is able to consistently resolve these dependencies to ensure a given bundle's code dependencies are satisfied so it can be used successfully. While this approach is certainly important and worthwhile, it does have one significant drawback.

Not all dependencies among bundles are related to code visibility. A prime example are bundles requiring specific capabilities from the underlying platform, such as particular hardware devices or windowing toolkit. Since the target of the dependency is not manifested as a bundle, there is no proper way to express such dependencies. It would be worthwhile if the framework and bundles could declare that they provide some arbitrary capability and other bundles could declare requirements on those capabilities.

This RFC investigates this issue and proposes a simple manifest-based metadata approach for declaring dependencies among bundles for reasons other than code visibility.

2 Application Domain

This RFC intends to provide a generic dependency model for bundles that will be understood and enforced by the resolver. Since existing resolve-time dependencies are related to code visibility management, the resolution of such dependencies and the `RESOLVED` bundle life cycle state have traditionally indicated that the bundles in question are “ready” to share and/or use their code. With a generic dependency model, this meaning is stretched to some degree.

Further, some dependencies are explicitly or implicitly tied to the `ACTIVE` bundle life cycle state and are not actually “ready” at resolve-time. For example, service dependencies require that both the provider and consumer components are `ACTIVE` for them to be “ready”, while extender/extendee dependencies typically require (at a minimum) the extender bundle to be `ACTIVE` for them to be “ready”. This RFC is intended to address “static dependencies” and not these sorts of “active dependencies”. However, since this is a generic dependency module, it is difficult, if not impossible, to limit its application to static dependencies. As such, this RFC will attempt to differentiate between static and active dependencies, but will only be able to enforce static (i.e., resolve-time dependencies).

The main impacts of modeling active dependencies with this mechanism are pushing the dependency resolution earlier in the bundle life cycle than what is strictly necessary and effectively concluding that the bundles are “ready” before they can technically be ready. The former impact cannot be avoided and typically results in having to deploy more bundles up front to resolve all dependencies at resolve time. The second impact can effectively be addressed by layering additional mechanisms on top to transition bundles to their appropriate states based on more detailed knowledge about their active dependencies.

3 Problem Description

While bundles are able to express a variety of code-related dependencies, not all bundle dependencies are related to code. Ultimately, this leads to situations where a bundle is unable to properly function, even though its code-related dependencies have been properly satisfied. Defining a standard way to express non-code-related dependencies will help eliminate such situations. The following use cases illustrate where such a mechanism could be applicable.

3.1 Operating environment dependencies (Bug 282)

It is often required to have one configuration of bundles run in multiple different operating environments. Here an operating environment is comprised largely of the OS, windowing system, locale, and processor architecture. In certain situations, such as on Unix systems that support GTK and Motif or when the user wants to run the same configuration on a different machine, these capabilities can vary greatly.

Currently, bundles are unable to state the operating environment in which they are valid, this is left to an external management agent to uninstall/reinstall bundles as appropriate. This sets up a need to communicate somehow with an agent on startup or requires managing multiple configurations. It also makes it difficult for provisioning agents to know which bundles are valid for which platforms.

3.2 Limitations of Bundle-RequiredExecutionEnvironment

3.2.1 Inability to capture java.* dependencies (Bug 390)

Sometimes bundle execution environment dependencies do not fall neatly into the defined set of execution environments. For example, consider the follow situations: a bundle that bundle limits itself to the APIs available in the Foundation 1.0 execution environment, but it finds it also needs the java.beans package. While it is possible to install a boot class path extension bundle to supply java.beans, the bundle in question has no way to indicate that it needs the extension bundle in the first place.

3.2.2 Lack of expressiveness (Bug 1000)

In some cases it may make sense to prohibit a bundle on a higher level VM. Currently, this is not possible with BREE because it specifies a list of EEs that the bundle is compatible with. For example:

```
    Bundle-RequiredExecutionEnvironment: J2SE-1.4
```

This means the bundle requires an EE that is compatible with J2SE-1.4. This includes J2SE-1.5, JavaSE-1.6 and all future VMs that will be compatible with J2SE-1.4. In some cases, it may make sense to provide a range limit to the EEs that the bundle can execute on. Something like this:

```
    Bundle-RequiredExecutionEnvironment: [J2SE-1.3, J2SE-1.5)
```

This would specify that the bundle can execute on an EE that is compatible with J2SE-1.3, J2SE-1.4 but can not execute on an EE that is compatible with J2SE-1.5 or higher.

3.3 Native code dependencies (Bug 1013)

The OSGi framework supports direct bundle dependencies on native code, but it is unable to deal with dependencies from one native library to another. While there is not a perfect solution to this issue, one approach to lessen the impact would be to devise a new system bundle fragment which could install native libraries in the LD_LIBRARY_PATH so that dependencies between native libraries could be resolved. However, a bundle would need some way to express a dependency on such native system bundle fragments if it were required for it to function properly.

3.4 Extender bundle dependencies

The extender pattern is a common OSGi pattern for providing some functionality or performing some task for unknowing bundles. An extender bundle listens for bundle lifecycle events, probes the associated bundles for metadata, then performs some activity on behalf of the bundle or some other purpose. The extendee bundle may have been written to explicitly expect the processing of the extender and may not be able to properly function without it. If the extendee bundle does not have a code dependency on the extender, then it has no way to express this requirement. Technically, this is an active dependency and can only be partially addressed by this RFC when modeled as a static dependency.

4 Requirements

REQ01	The solution MUST provide a way to capture dependencies among bundles not related to code visibility.
REQ02	The solution MUST provide a way to capture dependencies between bundles and the underlying platform/execution environment.
REQ03	The solution MUST be verifiable by the resolver without significant additional overhead.
REQ04	The solution MUST be limited to resolve-time enforcement/verification of dependencies.
REQ05	The solution MUST differentiate between static dependencies (i.e., resolve-time) versus active dependencies (i.e., active-time).

5 Technical Solution

This RFC proposes additional manifest headers for bundles to declare arbitrary provided capabilities and requirements on them. These headers impact resolution of the bundles involved and the dependencies among them, but have no other impact. To state this in another way using an analogy, while the resolver can be seen as injecting code visibility wires into a bundle or a service dependency framework can be seen as injecting services into a component, the mechanism proposed in this document only “injects” dependencies among bundles, but does not grant access to any capability as a result of this injection. It is predicated solely on the existence of declared capabilities to satisfy declared requirements and nothing more.

5.1 General approach

The general approach uses manifest-based syntax to allow a bundle to declare arbitrary capabilities and requirements. A capability is a set of attributes associated with a given namespace, where the namespace is simply a string name used to scope attribute names to avoid name clashes and convey semantics. For example, in OBR's use of generic capabilities and requirements, it uses the “package” and “bundle” namespaces to map package and bundle code dependencies, respectively.

The namespace syntax is limited to the same construction rules as bundle symbolic names (i.e., dot-delimited strings). The specification reserves the “osgi.*” namespace for its own purposes. Comparisons for namespaces and attribute names are case sensitive.

The approach for declaring a dependency uses the common OSGi manifest header syntax, since it avoids the need for new parsing code. To declare a capability, the `Provide-Capability` header is used, which has the following syntax:

```
Provide-Capability  ::= capability ( ',' capability )*
capability          ::= namespace ( ';' ( directive | attribute |
                                          typed-attribute ) )*
namespace           ::= symbolic-name
typed-attribute     ::= token ':' ( ( scalar-type '=' argument ) |
                         ( list-type '=' list-argument ) )
scalar-type         ::= 'String' | 'Version' | 'Long' | 'Double'
list-type           ::= 'List' ( '<' scalar-type '>' )?
list-argument       ::= '"' list-string ( ',' list-string )* '"'
list-string         ::= ( [^",\#x0D#x0A#x00] | '\"' | '\,' | '\\')*
```

Supported directives are "effective", with values of "resolve" and "active", and "uses", with value of a comma-separated list of package names.

The effective directive is used to indicate at which time the capability is in effect. The solution proposed by this RFC will only consider capabilities with effective time resolve; all other capabilities are ignored. If the effective directive is not specified, then resolve is assumed.

The uses directive for generic capabilities has the same meaning as defined for the `Export-Package` header. Specifically, it is a comma-delimited list of package names on which the associated generic capability depends. The specified package names are ones either exported or imported by the bundle providing the generic capability; if not, then it is ignored. It is not possible to specify uses constraints on arbitrary generic capabilities, only on packages.

The presence of the uses directive on a generic capability causes the resolver to place an additional constraint on any bundles requiring it. The additional constraint requires that if they also import a package used by the generic capability, then they must get it from the same source as the bundle providing the generic capability.

It is not possible to make generic uses constraints transitive. For example, with packages you can specify a uses constraints on your exported package to an imported package; doing so makes the uses constraints from the provider of the imported package apply to importers of the original exported package (i.e., they are transitive). The analog is not possible with generic capabilities. Uses constraints cannot be transitive across generic requirements on generic capabilities, since it is not possible to refer to generic requirements in the uses constraint of a generic capability.

An example capability is:

```
Provide-Capability: foo-extender; version:version="1.1.0"; secure=false
```

An untyped attribute defaults to String type.

An attribute of type `List` may also include a component type declaration as well as a comma-delimited list of values, such as:

```
versions:List<Version>="1.0.0,1.0.1"
```

If the component type is omitted, then it defaults to `String`. If the value of a list element must include quotes or commas, then they must be escaped with a backslash. If there is leading or trailing whitespace in the elements of the value string and the component type is `Long`, `Double`, or `Version`, then the whitespace is trimmed; otherwise it is significant.

To require a generic capability, the `Require-Capability` header is used, which has the following syntax:

```
Require-Capability  ::= requirement ( ',' requirement )*
requirement         ::= namespace ( ';' parameter )*
namespace           ::= symbolic-name
```

Supported directives are "effective", with values of "resolve" and "active", and "resolution", with values of “mandatory” and “optional”. The only supported attribute is “filter”. The “filter” attribute must be specified and its value must be a `quoted-string` containing a filter string (see 3.2.7 in spec) to match against provided capabilities. Since the filter attribute value is a `quoted-string` and the manifest parser will convert \\ to \, if the filter string needs to escape characters for the filter processor (for example ')'; see 3.2.7 in spec), then a double backslash will be need in the manifest to end up with a single backslash in the resulting filter string after manifest parsing.

The `effective` directive is used to indicate at which time the requirement is in effect. The solution proposed by this RFC will only enforce/verify requirements with effective time `resolve`; all other requirements are ignored. If the `effective` directive is not specified, then `resolve` is assumed.

The `resolution` directive is used to indicate whether a requirement is mandatory or optional, following the same semantics as optional imports.

An example requirement for some imaginary extender bundle is:

```
Require-Capability: foo-extender;
 filter="(&(version>=1.0.0)(!(version>=2.0.0))(secure=true))"
```

For this imaginary example, the requirement would not match the capability. Even though the extender is within the required version range, it does not provide a secure capability.

There is no concept of substitutable capabilities like we have for exported packages. Therefore, there is no special meaning placed on a bundle providing and requiring a given capability namespace, even if it matches itself. Such occurrences are treated as independently and are specifically allowed to match each other, which may be necessary if a fragment supplies something needed by a host.

5.2 Optionality

By default, all requirements are mandatory. However, `Require-Capability` does support the `resolution` directive which can be used to declare a requirement as optional.

5.3 Effective time

The default effective time of generic capabilities and requirements is at bundle resolve time, the same as for any of the existing code dependencies. The resolver algorithm only runs at resolve time, except in the cases of dynamically imported packages. In the case of generic capabilities and requirements, there does not appear to be an analogue to dynamically imported packages.

The resolver will only match a generic requirement to a capability if both have an effective time of `resolve`, since the framework ignores requirements and capabilities with any other effective time. On the other hand, upper layers are free to match `active` requirements to `resolve` capabilities.

5.4 Resolution result

The result of resolving generic capabilities and requirements is a set of wires. A wire connects a given requirement to its satisfying provider(s). The wiring result does not imply any sort of additional visibility or grant any new functionality between the provider and requirers. It does, however, tie their life cycles together like normal package wiring. This means if the provider is refreshed, then requiring bundles are refreshed as well.

RFC 151 is refactoring the Package Admin API and is providing an API for reflecting over bundle dependencies in a generic way, which will encompass reflecting over the resulting RFC 154 dependencies. Note that RFC 151 maps existing capabilities and requirements (e.g., packages, bundles, etc.) into this generic model for reflective purposes. As a result, it creates specific namespaces under “osgi.” that may be illegal to directly specify in the manifest. For example, it is not possible to use the “osgi.package” namespace to export or import a package. An installation exception will be thrown for any bundle specifying a non-allowed OSGi namespace in its manifest.

5.5 Multiple matching providers

If multiple providers match a given requirement, then similar rules as for multiple packages providers are followed. For example:

1. A resolved provider is preferred over an unresolved one.
2. Lower bundle identifier is preferred over a higher one.

A difference is that we do not assume anything about versioning for generic providers,

5.6 Cardinality of requirements

All generic requirements are assumed to match only a single provider.

5.7 Capability/Requirement verification

Unlike other headers (e.g., `Import-Package` and `Export-Package`) that have various verification rules associated with them (i.e., cannot duplicate or have certain attributes), generic capabilities/requirements have no such verification phase. So, for example, there is no way to say a bundle can only provide a given capability once or require overlapping providers.

5.8 Host/fragment issues

For code-related dependencies, fragment capabilities and requirements are merged with the host as long as they do not conflict. It is not clear how to detect a conflict for generic capabilities/requirements; thus, it is assumed that generic capabilities/requirements supplied by a fragment never conflict with those of a host.

Any fragment capabilities that are attached to a host bundle will appears as if they come from the host bundle after attachment. This is necessary since fragments can attach to multiple hosts, which effectively makes them different providers with respect to “uses” constraints.

6 Considered Alternatives

6.1 Discovery

OBR has demonstrated the potential usefulness of using generic requirements to aide in the discovery of interesting or related functionality. OBR refers to this as an “extends” relationship, which indicates that a given resource extends another target resource, although the target resource is unaware of it. Currently in OBR, this is simply represented as a requirement with a boolean “extends” flag. Having such a concept makes it possible for an OBR resolver implementation to proactively discover related resources and to implement policies around such extensions. It would be possible to support such a concept here with a directive on `Require-Capability`.

6.2 Uses constraints

It is not clear if “uses” constraints can be applied in a generic context.

6.3 Singleton capabilities

Yuck. We don't support them.

6.4 Mappings for existing headers

The existing bundle manifest headers could easily be mapped to this generic approach and in the case of OBR, they will be. Currently, the framework will not consider such replacements for the existing headers.

6.5 Bundle state related capabilities.

Framework level handling of bundle state issues was ruled out on complexity grounds. Whether it is possible and or desirable is an open question.

One option was to define a set of well known capability attributes that indicate the bundle state in which capability is available.

One option was to drop active state related capabilities from this specification altogether and make a statement that it should not be used to model these. The view seems to be that the informative resolution time error message makes modeling them in this way worthwhile.

6.6 Management agent control of namespaces

The effective time of all capabilities is bundle resolution time and this has some downsides for capabilities where bundle state matters.

Although addressing this in a standardized way at framework level would be complex, or at least premature, management agents that deal with bundle provisioning should not be prevented from innovating in this area.

For example, a management agent may know that a resolved extendee bundle has no need for a resolved extender bundle, and may also know that before the extendee is activated the extender should also be active.

In this case the management agent will want to take responsibility for handling the extender capability and would like the framework to ignore it. To this end the solution will make use of a framework property that tells the framework which capability namespaces it should ignore, as follows.

```
org.osgi.framework.processcapabilites = '-*'|*|(-)?namespace(,namespace)*
```

Where namespace is a generic capability namespace.

If the value is * then the framework will process all namespaces. This is the default.

In the absence of a leading '-', the list comprises the namespaces that the framework should process. All others should be ignored.

If the value is '-*' then the framework should ignore all namespaces.

If the value has a leading '-' then the listed namespaces should be ignored by the framework, but all others should be processed.

7 Security Considerations

The following permission is defined to control providing and requiring generic capabilities. For standard capabilities provided by the framework, the framework will need to implicitly grant bundle's the necessary CapabilityPermission to require those capabilities.

Class CapabilityPermission

org.osgi.framework

```
java.lang.Object
  └java.security.Permission
      └java.security.BasicPermission
          └org.osgi.framework.CapabilityPermission
```

All Implemented Interfaces:
Guard, Serializable

```
final public class CapabilityPermission
extends BasicPermission
```

A bundle's authority to provide or require a capability.

- The `provide` action allows a bundle to provide a capability matching the specified filter.
- The `require` action allows a bundle to require a capability matching the specified filter.

Since:
1.6
Version:
$Id: 11e47883a0c36de7cd69d0436287d41f8bd30f17 $
ThreadSafe

Field Summary		**Page**
static String	**PROVIDE** The action string `provide`.	14
static String	**REQUIRE** The action string `require`.	14

Constructor Summary	**Page**
CapabilityPermission(String name, String actions) Create a new CapabilityPermission.	14
CapabilityPermission(String namespace, Map<String,?> attributes, Bundle providingBundle, String actions) Creates a new requested `CapabilityPermission` object to be used by code that must perform `checkPermission` for the `require` action.	15

Method Summary		**Page**
boolean	**equals**(Object obj) Determines the equality of two CapabilityPermission objects.	16
String	**getActions**() Returns the canonical string representation of the actions.	15
int	**hashCode**() Returns the hash code value for this object.	16
boolean	**implies**(Permission p) Determines if a `CapabilityPermission` object "implies" the specified permission.	15
Permission Collection	**newPermissionCollection**() Returns a new `PermissionCollection` object for storing `CapabilityPermission` objects.	15

Field Detail

REQUIRE

```
public static final String REQUIRE = "require"
```

The action string `require`.

PROVIDE

```
public static final String PROVIDE = "provide"
```

The action string `provide`.

Constructor Detail

CapabilityPermission

```
public CapabilityPermission(String name,
                            String actions)
```

Create a new CapabilityPermission.

The name is specified as a dot-separated string. Wildcards may be used.

```
 name ::= <namespace> | <namespace ending in ".*"> | *
```

Examples:

```
 com.acme.capability.*
 org.foo.capability
 *
```

For the `require` action, the name can also be a filter expression. The filter gives access to the capability attributes as well as the following attributes:

- signer - A Distinguished Name chain used to sign the bundle providing the capability. Wildcards in a DN are not matched according to the filter string rules, but according to the rules defined for a DN chain.
- location - The location of the bundle providing the capability.
- id - The bundle ID of the bundle providing the capability.
- name - The symbolic name of the bundle providing the capability.
- capability.namespace - The namespace of the required capability.

Since the above attribute names may conflict with attribute names of a capability, you can prefix an attribute name with '@' in the filter expression to match against the capability attributes and not one of the above attributes. Filter attribute names are processed in a case sensitive manner.

There are two possible actions: `require` and `provide`. The `require` permission allows the owner of this permission to require a capability matching the attributes. The `provide` permission allows the bundle to provide a capability in the specified capability namespace.

Parameters:
`name` - The capability namespace or a filter over the attributes.
`actions` - `require`,`provide` (canonical order)
Throws:
`IllegalArgumentException` - If the specified name is a filter expression and either the specified action is not `require` or the filter has an invalid syntax.

CapabilityPermission

```
public CapabilityPermission(String namespace,
                            Map<String,?> attributes,
                            Bundle providingBundle,
                            String actions)
```

Creates a new requested `CapabilityPermission` object to be used by code that must perform `checkPermission` for the `require` action. `CapabilityPermission` objects created with this constructor cannot be added to a `CapabilityPermission` permission collection.

Parameters:
`namespace` - The requested capability namespace.
`attributes` - The requested capability attributes.
`providingBundle` - The bundle providing the requested capability.
`actions` - The action `require`.
Throws:
`IllegalArgumentException` - If the specified action is not `require` or any parameters are `null`.

Method Detail

implies

```
public boolean implies(Permission p)
```

Determines if a `CapabilityPermission` object "implies" the specified permission.

Overrides:
`implies` in class `BasicPermission`
Parameters:
`p` - The target permission to check.
Returns:
`true` if the specified permission is implied by this object; `false` otherwise.

---

getActions

```
public String getActions()
```

Returns the canonical string representation of the actions. Always returns present actions in the following order: `require`, `provide`.

Overrides:
`getActions` in class `BasicPermission`
Returns:
The canonical string representation of the actions.

---

newPermissionCollection

```
public PermissionCollection newPermissionCollection()
```

Returns a new `PermissionCollection` object for storing `CapabilityPermission` objects.

Overrides:
`newPermissionCollection` in class `BasicPermission`
Returns:
A new `PermissionCollection` object suitable for storing `CapabilityPermission` objects.

---

equals

```
public boolean equals(Object obj)
```

Determines the equality of two CapabilityPermission objects. Checks that specified object has the same name and action as this `CapabilityPermission`.

Overrides:
`equals` in class `BasicPermission`
Parameters:
`obj` - The object to test for equality.
Returns:
true if obj is a `CapabilityPermission`, and has the same name and actions as this `CapabilityPermission` object; `false` otherwise.

hashCode

```
public int hashCode()
```

Returns the hash code value for this object.

Overrides:
`hashCode` in class `BasicPermission`
Returns:
Hash code value for this object.

8 Document Support

8.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

8.2 Author’s Address

Name	Richard S. Hall
Company	
Address	
Voice	
e-mail	richard.s.hall@oracle.com

8.3 Acronyms and Abbreviations

8.4 End of Document