for *namespace* and *class*, and there are rules for determining the appropriate *version*.

The recipe *name* must always be specified.

For *external references* within a recipe, the default *class* is the *class* of the recipe making the reference. The proper *class* is determined when the recipe is *linked* (see Section 3.2.4.1.3).

Where a recipe *version* is unspecified, the recipe of that *class* and *name* having the highest *approval level* will be used. Where several recipes are found with the same *approval level*, the one with the highest *version* will be used. Where the *version* is unspecified in an *external reference*, the correct *version* is determined when a **linked recipe set** is built by the **link** operation.

For references with *managed recipes*, if no *namespace* is specified, the default is *namespace* of the referring recipe.

3.2.4 *Advanced Recipe Capabilities*

3.2.4.1 *Multi-Part Recipes* — Use of multi-part recipes is often dictated by the *recipe executor*, which may require a set of recipes of different *classes* for its process. In addition, support for multi-part recipes can be helpful for the user, in particular where recipes are otherwise long. The recipe language may define use of subrecipes analogous to that of subroutines in ordinary programming languages. This enables better re-use where a high degree of similarity exists between different recipes.

Where recipes of more than one *class* are required, it is convenient to allow one recipe to reference another. A recipe in the PROCESS *class*, for example, may refer to recipes in other *classes* to ensure that an entire set of recipes is *executed* together.

It is also convenient to allow one recipe to reference other recipes within the same *class*, where a substantial set of instructions and/or *settings* are common to a variety of other recipes.

A reference within the *body* of one recipe to a different recipe is called an **external reference.** The exact syntax in which this is done is determined by the recipe language used. An *external reference* may also specify values for *variable parameters* defined in the referenced recipe or in one of its *subrecipes*.

3.2.4.1.1 *Subrecipes* — When one recipe references another recipe, both recipes are required for *execution*. The starting recipe is called the **main** recipe.

Subrecipes are those recipes that are referenced by the *main* recipe, or by another *subrecipe* of the main recipe. A recipe that references other recipes is a **parent** recipe to the referenced *subrecipes*. A parent recipe is not necessarily a *main* recipe. A main recipe must not be referenced by any of its *subrecipes*.

External references to recipes within the same *namespace* may or may not be explicit. That is, references to the current *class* (i.e., the *class* of the recipe being *verified*) may be implicit, and *versions* may be left to the rules for determining the default *version* at *link-time*.

A recipe may also reference recipes in another *namespace*. This occurs either when a recipe is to be **delegated** or when a recipe in a *namespace* accessed by multiple *agents* must reference a hardware-dependent or other *agent-specific* recipe kept in a **default namespace**. The *default namespace* of a *recipe executor* is always referenced within a recipe as the namespace named "Default". This name is interpreted by the *recipe executor* at execution time, based on its attribute DefaultNamespace (see Section 6.3).

When specified in a text string within a recipe with the subrecipe's *identifier*, the *identifier* is preceded by the value of the *namespace's* identifier, followed by a greater-than symbol ">". This is called the **recipe specifier**. For a recipe ETCH version 5 in a *namespace* named "NS-MOM", this would be specified as

"NS-MOM>/PROCESS/ETCH;5".

According to the rules of OSS, object type may be omitted when it may be otherwise determined. In RMS, determination is made according to order.

3.2.4.1.2 *Delegated Recipes* — A recipe may reference recipes in another *namespace* that are to be executed by a *component agent* and are termed **delegated recipes**. Although referenced by another recipe, *delegated recipes* are not considered as *subrecipes* of the *parent* (referencing) recipe as they are to be *executed* by a difference *agent* and are themselves required to be a *main* recipe. A reference to a *delegated recipe* is the equivalent to an automated "select" and "start" sequence of commands from the *supervisor*. As such, they are generally subject to the constraints of a *select* operation.

Delegated recipes may or may not require the *name* of the *executing agent* (*recipe executor*), depending on the rules of the particular implementation. For example, a *supervisor* may be able to determine the appropriate *executing agent* at run-time, based on the *namespace* specified. However, the ability to specify a particular *executing agent* is important to the user, as process results may be sensitive to a particular equipment installation. Therefore, recipe languages that support delegation shall allow specification of the particular *executing agent* to be used. The text format of the full **recipe specifier** in this case is:

"Agent-Name>Namespace-Name>CLASS/Name;Version".

The required order of *identifiers* within the *specifier* is: agent, *namespace*, recipe.

3.2.4.1.3 *Linked Recipe Sets* — Where one recipe references another, a set of recipes to be executed together is formed by starting at the *main* recipe and collecting the *external references* to identify all the members of the set. This set is called a **linked recipe set**, and the operation of collecting the references is the **link** operation.

3.2.4.2 *Variable Parameters* — **Variable parameters** are variables that can be assigned values from outside the recipe itself. *Variable parameters* allow recipes to use variables rather than constants for such actions as setting temperature setpoints, time delay intervals, and data-set names. This capability greatly extends the re-usability of a recipe.

A *parameter* first is formally **defined** within a recipe body and given a unique **parameter name**, a **parameter initial value** (default value) for use when the recipe is *selected* for execution, unless overridden. Where applicable, the *definition* also includes a **parameter restriction** that represents one or more conditions that any *value* assigned to that *parameter* is required to satisfy to be valid.

The syntax for *parameter definitions* in recipe *bodies* remains unspecified, so long as it conforms to the syntax of the recipe language and contains the required elements. Other restrictions concerning the use of *variable parameters* may be imposed by the *recipe executor's* supplier through the specification of the recipe language.

The **parameter domain** is the set of all possible values of a given *form* (see Section 1.4.3) that fulfill the conditions of the **parameter restriction** (if any).

The form need not be included in the *parameter definition* declared within the recipe *body*, but rather may be derived by the *recipe executor* during the *verification* process from the way in which the *parameter* is applied within the recipe. If temperature is maintained internally as an unsigned integer, for example, those *parameters* that are used in the recipe to assign temperature setpoints and ranges would typically be required by the supplier of the *executing agent* to represent unsigned integers as well. The supplier shall document the forms and valid *domains* for each of the *parameters* that may be used as a *variable parameter*.

NOTE: The *parameter's value* may be changed from its *initial* (default) *value* within the text of a recipe, and it may also be set as an "argument" passed to a *subrecipe*. These capabilities are a function of the recipe language and are beyond the scope of RMS.

Two categories of *parameters* are defined: **numeric** and **non-numeric**.

3.2.4.2.1 *Numeric Parameters — Numeric parameters* include all *parameters* that can take on *any numeric value* for its *format type* between a **parameter low limit** and **parameter high limit**.

The *parameter restriction* for a *numeric parameter* in any of the attributes that store *parameter definitions* shall be a text string that conforms to one of the following:

- *"(a,b)␣UNITS" the* domain *of numbers x such that* $a<x<b$
- *"(a,)␣UNITS" the* domain *of numbers x such that* $a<x<+\infty$
- *"(,b)␣UNITS" the* domain *of numbers x such that* $-\infty<x<b$
- *"UNITS" the* domain *of numbers x such that* $-\infty<x<+\infty$.

where:

1. a and b are *numeric values* formed from the digits "0" through "9", the plus and minus signs "+" and "-", the period ".", and the letters "E" or "e" for floating point numbers,
2. ␣ represents optional *whitespace*, and
3. UNITS is a valid case-sensitive string conforming to the Units of Measure Identifiers (see SEMI E5, Section 9).

For example, both of the following strings are valid: "(500,2000)␣degC" and "(0,100)".

To include equality, parentheses are replaced by the square brace "[" and/or "]" on the left or right end, respectively. For example, "[a,b)" represents the domain of numbers x such that $a\leq x<b$.

To specify units only, with no restriction on range, the *low limit* may be set to $-\infty$ and the *high limit* to $+\infty$. However, the *low* and *high* limit shall not exceed absolute minimum and maximum limits set for that *parameter* by the *recipe executor's* supplier.

Numeric parameters with no *parameter restriction* are pure numbers (with no units) with a domain of $(-\infty,\infty)$.

A *numeric parameter* shall have a *format type* of one of the following:

- *text string of a length of no more than 80 characters,*
- *signed or unsigned integer of an even length,*
- *floating point number of an even length.*

Text strings are restricted to the character set defined above to represent the *low* and *high limits* of the *domain* and shall convert to a *numeric value* within that *domain*.

3.2.4.2.2 *Non-Numeric Parameters* — A **non-numeric parameter** is any parameter other than a *numeric parameter*, including *parameters* whose *domains* are sets of discrete numbers that cannot be represented by a single mathematical interval, and strings that represent names. *Non-numeric values* may or may not place restrictions on the replacement *value*. A *parameter* that contains the name of a wafer map, for example, cannot be easily restricted by a general rule[7], whereas a *parameter* that contains a string identifying thermocouple type can be restricted to one of a defined, unordered set of valid strings. Restrictions for *non-numeric parameters* might also consist of, or include, logical expressions that shall evaluate to TRUE before a value may be used.

If a recipe language has been defined for the corresponding *class* of recipes, the syntax for the *restriction* is taken directly from the formal *definition* of that *parameter* within the body of a *source form* recipe (and therefore conforms to the specification for that recipe language). Otherwise, the restriction is expressed in a syntax defined by the *recipe executor's* supplier specifically for *parameter* attributes.

For example, the *parameter* for a furnace recipe may specify which set of PID *values* should be used, where sets A-F exist on the *equipment*. The automatic *restriction* on the *value* for such a parameter, as specified by the *recipe executor's* supplier, might be "{A,B,C,D,E,F}" with the default for the *value* specified as the character "C" with a *format type* of "ASCII string of length 1".

In another example, a *restriction* might be that the *value* be taken from the enumerated set

"{cassette, lot , batch }".

3.2.4.2.3 *Agent-Specific Parameters* — In order to "tune" a recipe so that it produces the same result on all *recipe executors* of the same type, it may be necessary to provide a different *initial value* for the *parameter* or a different *parameter restriction* for individual *executing agents*. A special editing facility may be provided by the *recipe manager* to allow the original *value* and/or *restriction* of a *parameter* to be modified for a specific *executing agent*.

NOTE: *Agent-specific parameter definitions* replace the corresponding original *definitions* in the recipe that is *downloaded* to the *recipe executor*. At run-time, they are superseded by *parameter values* specified for the *recipe executor's select* operation.

3.2.5 *Attributes* — The name of an attribute is a text string required to be unique for its object. The names of the attributes defined in this document are reserved for their standard use.

Identification attributes are those used to identify the object in OSS. These attributes are available through OSS, but in RMS, services are handled separately from the other attributes and are not transferred in parameters for "recipe attributes".

Attributes other than *identification* attributes and **mandatory** attributes discussed below always have a defined **default value**. Attributes with a *default value* not otherwise specified are considered to have a **null value** corresponding to their form as their *default value*. The *null value* for a text string is a zero-length string. The *null value* for a numeric form is zero. The *null value* for a boolean form is FALSE. The null value for an empty list or structure is an empty (zero-length) list. The manner of representing *null values* is left to the protocol.

An attribute is **cleared** or **reset** by setting its value to the *default value*.

Mandatory attributes are attributes that are required to always have a *non-default value*.

Certain attributes are **required**, including all *mandatory attributes*. A **required** attribute is one that shall be supported with a *non-default value*. For example, the BodyFormat attribute is *required*. This means that *object form* recipes (for which BodyFormat has a *non-default value*) shall be supported by a *namespace*.

Any attribute defined in this standard shall only be modified according to the rules specified for that attribute.

Attributes of the *managed recipe* and the *execution recipe* may be accessed through Object Services. However, only certain of the attributes may be set through Object Services.

For purposes of OSS, all attributes defined in RMS shall be *recognized*. That is, a response to a GetAttr service request that references any attribute of an object defined in RMS for that object shall not return an "invalid attribute" error. If the attribute is not supported by the application, then it shall show the value of that attribute as having the null value appropriate for its form, and it shall deny attempts to set its value through the SetAttr service.

7 Typically, the validity of parameters which are unrestricted names can only be established at run-time.

3.2.5.1 *Descriptors* — Two types of *mandatory* attributes of particular importance are the **timestamp** and **length** attributes. These are provided for the *body*, for each *agent-specific dataset*, and for the set of attributes of the recipe object itself, including the *timestamp* and length attributes of the *body*.

A **descriptor** contains the *length* and *timestamp* attributes (in that order) of one or more of the aspects of a recipe: its attributes, its *body*, or an *agent-specific dataset*. *Descriptors* are used to compare two recipes stored in different *namespaces* or in a *namespace* and in the *recipe execution area* of a *recipe executor*.

NOTE: Provision of a real-time clock or other device capable of counting time in centiseconds greatly improves the value of the *timestamps* for this comparison.

3.3 *Full and Minimal Recipe Models* — Figure 3.5 contains an object model for the *managed* and *execution* recipes supporting all of the standard *attributes* defined in RMS. *Non-identifier* attributes are shown in alphabetical order. Many of these attributes are not required for minimal applications. Minimal models for the *managed* and *execution* recipes are shown in Sections 3.4.2.3 and 3.5.2, respectively.

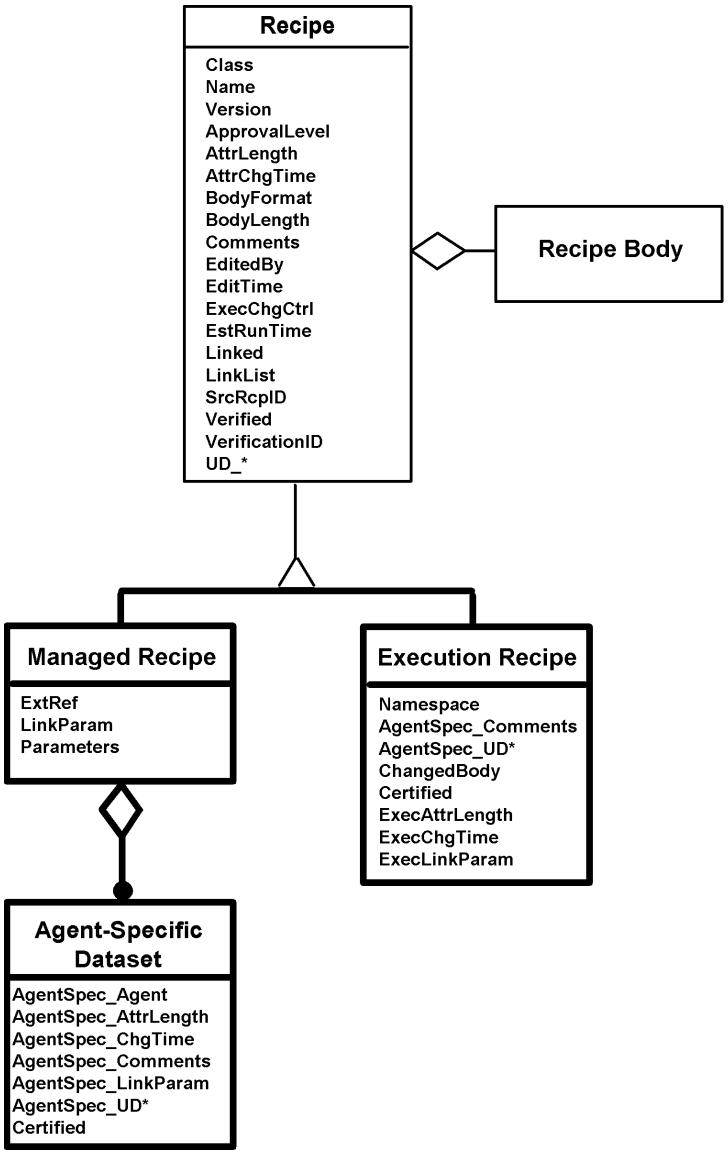


Figure 3.5
Full Recipe Object Model

3.4 *Managed Recipes* — This section defines the *managed recipe* and its attributes.

3.4.1 *Generic Attributes* — For clarity, the attributes of the *managed recipe* are sometimes called **generic** attributes to distinguish them from *agent-specific* attributes or attributes of the *execution recipe*.

Attributes that are not specifically defined in this standard may be defined by an application or by the user. Attributes that are defined by a *namespace* or by a *recipe executor* for its own use shall be changed only by rules specified and documented by the originator of the attribute.

To prevent conflict between non-standard attribute names and names which may become standard in the future, non-standard attribute names shall start with the prefix "UD_" (the two characters "U" and "D" followed by the underline character). It is the responsibility of the originator to provide unique attribute names. When using other RMS services, non-standard attributes shall follow all standard attributes.

It is desirable that a method be provided for the user to define new attributes.

3.4.1.1 *Timestamp Attributes* — **Timestamp** information (date and time of the last change) is important to recipe management. *Timestamp* attributes are maintained by the *recipe namespace* and may not be changed otherwise.

A **timestamp** attribute shall always contain the date and time that the particular aspect of the recipe was last changed. This attribute is a text string of the form "yyyymmddhhmmsscc" for the year yyyy, the month mm, the day dd, the hour hh, the minutes mm, the seconds ss, and the centiseconds cc.

The *timestamp* of the *body* is called EditTime and is set when a recipe is first **created** and updated whenever the *body* is modified in any way.

Because a recipe's attributes may be changed without changing the body, both the recipe's *generic* attributes and each set of *agent-specific* attributes themselves each have a *timestamp* attribute. For the *generic* attributes, the *timestamp* attribute is called AttrChgTime. The set of *agent-specific* attributes (of an *agent-specific dataset*) has its own *timestamp* attribute, called AgentSpec_ChgTime.

3.4.1.2 *Length Attributes* — There are three **length attributes**, one which contains the length of body, one that contains the length of the *generic* attributes, and one that contains the length of the *agent-specific* attributes. *Length* attributes are calculated without regard to either protocol overhead or storage overhead, such as proprietary formats used for internal storage that may change from one implementation to another. This preserves the length across different conventions used for recipe storage and different communications protocol.

The *length attribute* of the *body*, BodyLength, contains the length of the *body* in bytes. The *length* of the *body* is calculated as the number of bytes it will require when transferred, excluding any overhead, such as that which might be required for protocol format information.

The length of an individual attribute is calculated as the sum of the lengths of the attribute name and the attribute value.

The length of a set of attributes shall be calculated as the sum of the lengths of the individual attributes that are set to a *non-default value* at the time the calculation is performed, including the *length attribute* itself. All attributes set to their *default value*, at the time the *length* is calculated, are excluded from the calculation. This is because only the attributes with a *non-default* value are transferred when a recipe is moved into or out of a *namespace*. It also results in a more significant change to value of the *length* attribute when an attribute is set to a non-default value.

3.4.1.3 *Descriptors* — The *descriptors* of the *managed recipe* are the **body descriptor** (BodyLength and EditTime), the *generic* **attribute descriptor** (AttrLength and AttrChgTime), and the *agent-specific descriptor* (AgentSpec_AttrLength and AgentSpec_ChgTime). The **recipe descriptor** for a *managed recipe* consists of the *attribute descriptor*, the *body descriptor* (in that order), followed by *descriptors* of any existing *agent-specific datasets*.

3.4.2 *Managed Recipe Object Attribute Definitions* — This section provides the formal definitions for the recipe's *generic* attributes. For Object Services, a recipe is considered as the owner of its *components*, any *agent-specific datasets*. The *body* may not be accessed through Object Services.

Attributes in Tables 3.1 and 3.2 are presented in the following order:

- identification *attributes, including object type and identifier, appearing above the heavy line in Tables 3.1 and 3.2,*
- mandatory *attributes and the other required attributes, in the order in which they are to appear when a recipe is transferred,*
- *optional attributes in alphabetical order, and*
- *non-standard attributes, which are transferred last.*

When transferring a recipe with RMS services, *identification* attributes are not included in the list of attributes, and required attributes are sent, in order, before optional attributes. *Non-mandatory* attributes having their *default value* are not transferred, since their absence indicates their value, but they are always available through Object Services. Order of optional attributes is not dictated.

3.4.2.1 *Generic Attribute Definitions* — Table 3.1 provides the formal definition of the *generic* attributes of the *managed recipe*.

Table 3.1 Managed Recipe Attribute Definition

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*
Identification Attributes				
ObjType	The object type.	RO	Y	Text: "MRcp"
ObjID	An identifier derived from Class, Name, and Version. No part of a recipe's identifier shall be changed except through *renaming*.	RO	Y	Formatted text.
Name	A logical name assigned by the user when the recipe is *created* or *renamed*.	RO	Y	Text.
Class	The recipe's class (e.g., "/PROCESS/" or "/PROCESS/LOADER/").	RO	Y	Formatted text: "CLASS/CLASS/../CLASS/"
Version	The version of the recipe.	RO	Y	Text.
Mandatory Attributes				
AttrLength	The total length of the *generic* attributes, in bytes. *Mandatory*.	RO	Y	Unsigned integer.
AttrChgTime	Timestamp of the last change to a *generic* attribute. *Mandatory*.	RO	Y	Formatted text.
BodyLength	Length of the recipe's body, in bytes. *Mandatory*.	RO	Y	Unsigned integer.
EditTime	Timestamp of when the *body* was *created* or last *updated*. *Mandatory.*	RO	Y	Formatted text. *Timestamp* format.
Required Attributes				
BodyFormat	Indicates the form and format of the recipe's *body*. *Default* is zero.	RO	Y	Enumerated unsigned integer: 0 = *source*, 1 = *object*, > 1 reserved.
Verified	Indicates whether the recipe's body is syntactically correct. Reset when the recipe is *created* or *updated*. Default is FALSE.	RO	Y	Boolean.
Linked	Indicates whether the recipe is linked. Reset when the recipe is *originated*, *verified*, or *unlinked*. Default is FALSE.	RO	Y	Boolean.
Optional Attributes				
ApprovalLevel	Indicates the level of approval assigned by an *authorized user*. Default is zero. Reset when the recipe is *originated* or *linked*. For a *linked* recipe, may not be higher than any of its *subrecipes*.	RW	N	Unsigned integer.
Comments	User comments.	RW	N	Text. Maximum length is 80 characters.
EditedBy	The name of the person who last edited the recipe.	RO	N	Text. Maximum length is 40 characters.
EstRunTime	The nominal or estimated execution (run) time of the recipe, in seconds. Reset when the recipe is *created* or *updated*. Set when the recipe is *verified*. May be	RW	N	Unsigned integer.

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*
	recalculated to total time for a main recipe when *linked*. Used for scheduling purposes. Algorithm for calculation shall be documented. Default is 0.			
ExecChgCtrl	Specifies change control requirements for recipe. (See Section 6.5.)	RW	N	Binary. Bitwise (MSB = 8**). 1 - The recipe body may be changed, 2 - Change notification is required, 3 - Recipe may be selected after change, 4 - Most recent parameter settings shall be saved.
ExtRef	A list of all recipe *specifiers* as referenced within the recipe. Explicit *versions* not required. Reset when the recipe is *created*, *updated*, and *verified*.	RO	N	List of formatted text.
LinkList	A complete list of recipe *specifiers* found in the ExtRef attribute of a *main* recipe and all of its *sub-recipes*, with duplicates removed and all *versions* explicitly determined. Set for the *main* recipe when linked. Reset when the recipe is *originated* or *verified*. Required for multi-part recipe support.	RO	N	List of formatted text.
LinkParam	A list of all variable parameter definitions contained in the Parameters attribute of a *main* recipe and all of its *subrecipes*, with duplicates removed. Reset when the recipe is *created*, *updated*, or *verified*. Set when the recipe is *linked*. Required for *variable parameter* support.	RO	N	Structure composed of parameter name, initial value, and restrictions.
Parameters	A list of variable parameter definitions contained in the recipe. Reset when the recipe is *created*, *updated*, and *verified*. Set when the recipe is *verified*. Required only for *variable parameter* support.	RO	N	Structure composed of parameter name, initial value, and restrictions.
SrcRcpID	*Identifier* of the *source form* recipe from which a *derived object form* recipe is derived. Value determined by the *verifier* of the recipe. Required only for support of *derived object form* recipes.	RO	N	Formatted text.
VerificationID	Identification code set by the *verifier* of the recipe. May be used to determine out-of-date formats that need to be *reverified*.	RO	N	Text. Maximum length is 40 characters.
UD_*	Non-standard attribute defined by supplier or *user*. Asterisk indicates the part of the attribute name that is provided in this definition. Shall be preserved exactly except by the entity that defined it.	RO	N	Varies with definition. Text form is limited to 80 characters.

** NOTE: SEMI E4 and E5 number bits 1–8, where Bit 8 = MSB (most significant bit).

3.4.2.2 *Agent-Specific Attribute Definitions* — The names of user-defined attributes shall start with the prefix "AgentSpec_UD_" and shall be preserved without modification from transferred recipes. These attributes follow all standard attributes when transferred.

An *agent-specific dataset* exists only if an *agent-specific* attribute other than the *timestamp* and *attribute-length attributes* of the *dataset* itself have been set.

Table 3.2 defines the attributes of the *agent-specific dataset object*.

Table 3.2 Agent-Specific Dataset Object Attribute Definition

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "MRcpASDS"
ObjID	The object's identifier. Contains the value in AgentSpec_Agent.	RO	Y	Text.
AgentSpec_Agent	The name of the *executing agent* to which the other attributes in the *dataset* apply. *Mandatory*.	RO	Y	Text.
AgentSpec_AttrLength	The length of the *agent-specific* attributes, in bytes. *Mandatory*.	RO	Y	Unsigned integer.
AgentSpec_ChgTime	Timestamp of when an *agent-specific* attribute was last changed. *Mandatory*.	RO	Y	Formatted text.
AgentSpec_Comments	Comments specific to the *agent* entered by the author.	RW	N	Text. Maximum length is 80 characters.
AgentSpec_LinkParam	A list of *variable parameter definitions* modified from the list in LinkParam. Valid only for a *linked main* recipe. *Parameter name* and form may not be changed.	RO	N	List of Structure composed parameter name, value, and restrictions.
Certified	The certification-level for the specific *agent*, assigned by an *authorized user*. Reset when AgentSpec_LinkParam is modified. Required for *certification* support.	RW	N	Unsigned integer.
AgentSpec_UD_*	Non-standard attribute defined by the supplier or user. Asterisk indicates the part of the attribute name that is provided in this definition. Must be preserved exactly except by the defining entity.	RO	N	Varies with definition. Text form is limited to 80 characters.

3.4.2.3 *Minimal Managed Recipe* — The minimal model for a *managed recipe* is shown in Figure 3.6. Only required attributes are supported in this model. This model can only be used for single-part recipes in the dedicated *namespace,* such as the *default namespace* required for stand-alone *equipment*. There are no *agent-specific datasets*.

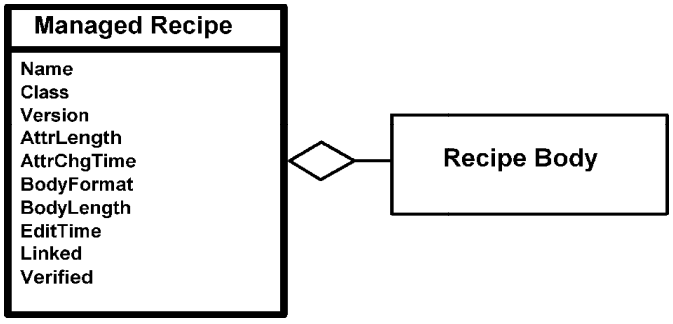


Figure 3.6
Object Model for Minimal *Managed* Recipe

3.5 *Execution Recipes* — The type of recipe handled by a *recipe executor* is called an *execution recipe*. This type of recipe has attributes and a *body* but does not have an *agent-specific dataset*. Most of the *generic* attributes of the *managed recipe*, and certain of the *agent-specific* attributes, are also attributes of the *execution recipe*, while in other cases, a *generic* attribute and an *agent-specific* attribute are merged and placed in a new *execution recipe* attribute.

A *managed recipe* is converted to an *execution recipe* when the recipe is *downloaded* to the *recipe executor*, and correspondingly, an *execution recipe* is converted to a *managed recipe* when *uploaded* from the *recipe executor*.

The *execution recipe* is defined in detail in Section 6.3.

4 Recipe Namespace

This section defines the basic conceptual model for the **recipe namespace**, a logical domain for recipe storage and management, within which the *identifier* of a recipe is guaranteed to be unique. The model in the current section is applicable to implementations with centralized storage. Section 5 defines an extension of the basic model for a distributed recipe namespace.

In general, the term "namespace" refers to a domain of unique *identifiers*. The issue of *namespace* boundaries exists for all object types, particularly in a distributed environment, and is not unique to recipes. However, within the context of RMS, **namespace** is used as a synonym for *recipe namespace*.

4.1 *Motivations* — A recipe's *identifier* may not be unique across different *namespaces* or throughout a factory. (That is, a given identifier may be used by internally different recipes except within a single *namespace*.) A primary role of a *namespace* is to define an area within which the uniqueness of any given recipe *identifier* may be guaranteed.

The requirements for Recipe Management that are addressed in this section include:

- *to define the boundaries of specific areas where recipes may be uniquely identified, stored, and retrieved,*
- *to define the* attributes *and operations for the management domain of recipes, including recipe protection and recipe operations,*
- *to allow control of removable media,*
- *to allow stand-alone* equipment *to* execute *recipes,*
- *to facilitate smooth integration of stand-alone* equipment *into on-line factory systems,*
- *to allow integrated* equipment *to continue to execute recipes when communications have been lost, and*
- *to provide the basis for the* ***distributed*** *model in Section 5, which allows a* supervisor *to use and manage the storage capacity of its* subordinate agents.

4.2 *Namespace Model* — The *namespace* model serves two major purposes. First, it provides a common set of management rules. Second, it allows a set of recipes to be shared among a group of *executing agents* that have a common process type, common functionality, and a common recipe language.

Figure 4.1 shows the basic *namespace* model with four objects: the recipe, the *namespace* itself, a *namespace* component called **recipe namespace segment**, and a **recipe namespace manager**.

The **recipe namespace manager** (or **manager**) represents the interface for the *namespace* to the external world and the internal decision authority within the *namespace*. All services for the *namespace* and its recipes are provided by the *manager*. The *namespace* itself is passive. While it has important *attributes*, the *namespace* has no operations, and it provides no services.

The **recipe namespace segment** (or **segment**) represents both the internal storage element and the actual manipulation of recipes within the *namespace*, under the supervision of the *manager*. In the basic model, it also provides no public services. The relationship between the *segment* and the *manager* in this model is presumed to be internal to an application.

Figure 4.1 represents a *namespace* with centralized storage. However, it can easily be extended to the **distributed recipe namespace**, which may have multiple *segments*, each provided by a different external *agent*. For that case, the *manager* and *segment* must provide public services for one another.

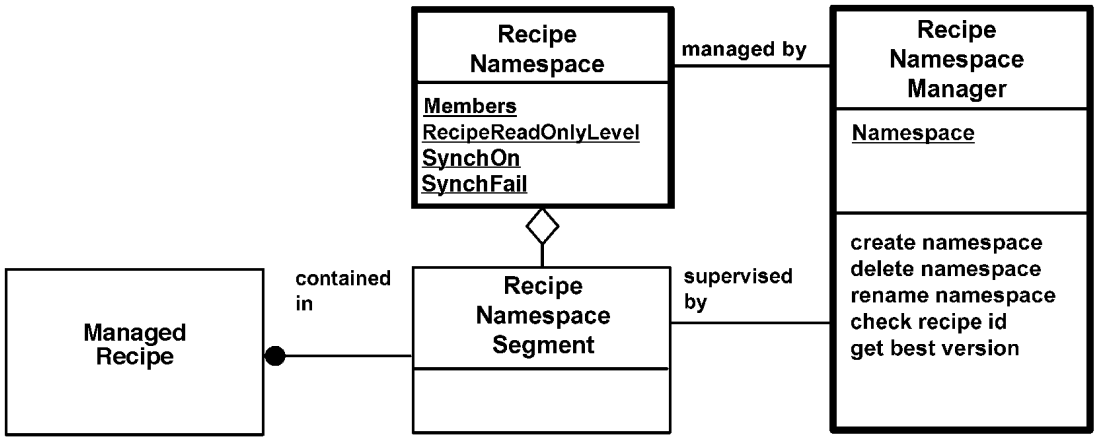


Figure 4.1

Recipe Namespace Model

All operations performed on a recipe through Recipe Management *services* are performed within the *namespace* by, or under the supervision of, the **namespace manager**, with two exceptions. A *namespace manager* is not required to understand the syntax of the recipe languages. For this reason, recipe **editing** (modifying a recipe's *body*) is not part of the *namespace* definition, and *verification* shall be provided by a **recipe executor** of an appropriate *executing agent* (see Section 6). These two activities require an in-depth understanding of the particular recipe languages used. Their separation from the other activities allows generic *namespace* capabilities to be provided.

4.3 *Namespace Specifications* — The combination of the *recipe namespace*, its *segment* component, and its *manager* provides storage, retrieval, and management of recipes conforming to RMS.

A *namespace manager* is responsible for maintaining the integrity of the *namespace*, the integrity of the recipes within the *namespace*, and the integrity of the recipe *identifiers*. It understands the rules regarding recipes, their *attributes*, and their components, and it is responsible for enforcing those rules. It will not allow a *read-only* recipe to be changed or deleted, for example. Therefore, it will not accept a recipe with an *identifier* already used by a *read-only* recipe.

A *namespace* has no restrictions on the **read access** of a recipe as a whole or of its *attributes*, nor is it concerned with the uses to which they might be put outside the *namespace*. The ability for multiple *agents* to share the same recipes is determined solely by the ability of the *agents* to access the same *namespace*. Issues of security and authentication are beyond the scope of RMS.

The term *recipe namespace*, or *namespace*, is used inclusively to refer to those *attributes*, operations, and other requirements common to both the *namespace* described in this section and to its *distributed* subtype. The term **centralized namespace** is used in references to an instance (implementation) of the basic model and to clarify statements that do not apply to instances of the *distributed* subtype.

A *centralized namespace* is analogous to a single directory of files, where duplicate file names are not allowed. It may exist in several different configurations that are incidental to *namespace* requirements.

It is possible to provide a *namespace* that uses removable media for its physical storage. Recipes may then be transferred to and from this *namespace* to any other *namespace*.

In no case shall it be possible to transfer recipes into a *namespace* except according to the requirements for *namespace management*. For example, a *read-only* recipe may not be replaced.

The *centralized namespace* may be applied several ways. An *agent* with *execution* capabilities that may be operated in stand-alone mode shall provide itself with a *centralized namespace* to be used when operating in stand-alone mode. Other *agents*, such as diskless process modules in a cluster, may expect to rely upon the *namespace* capabilities provided by the cluster *supervisor*.

4.4 *Member Agents* — A *namespace* that contains recipes used by multiple *agents* is called **shared**. Otherwise, it is called **dedicated** or **non-shared**. Figure 4.2 illustrates two *dedicated namespaces*, one provided by an etcher and one by its host. This terminology is introduced for clarification and descriptive purposes only.

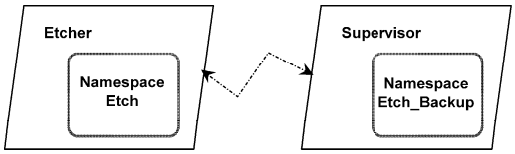


Figure 4.2

Host Backup of Equipment Namespace

The Members attribute of the *namespace* contains the names (object identifiers) of the *agents* that use the recipes in the *namespace*, and these *agents* are called **member agents** (**members**) of the *namespace*. The *namespace manager* uses this information as it may require assistance from a *member agent* to *verify* a recipe.

If the Members attribute of the *namespace* contains multiple *agent* names, then the *namespace* is *shared*. Otherwise, it may be *non-shared* or not yet completely set up. If it is empty (null), the *namespace* has not been completely set up and is not fully functional, as certain operations, including recipe *verification*, require it to have content.

4.5 *Illustrations* — This section provides illustrations of several possible configurations of a *centralized namespace*.

Figure 4.2 illustrates an etcher that has its own local *namespace* and that communicates with its *supervisor* over an RS-232 line. The *supervisor* in this example maintains a separate *namespace*, providing backup for recipes of particular significance.

Figure 4.3 illustrates a *supervisor* with four diskless subordinates. In this configuration, the *supervisor* provides recipes for all four subordinates from a *centralized namespace*.

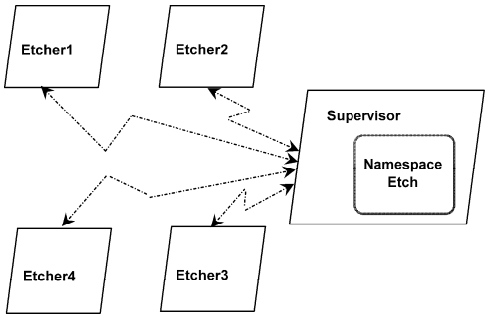


Figure 4.3

Namespace Provided by Supervisor

Figure 4.4 illustrates four *executing agents* on a common network with a *supervisory agent* and a sixth *agent* providing a *centralized namespace*. Each *executing agent* is able to access the *namespace* independently of the *supervisor*.

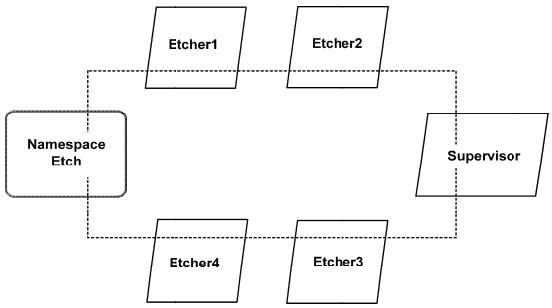


Figure 4.4

Shared Network Access

Figures 4.3 and 4.4 both represent examples of *shared centralized namespaces*.

4.6 *Attribute Definition Tables* — Table 4.1 defines the attributes of the recipe namespace.

Table 4.1 Recipe Namespace Attribute Definition

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RNS"
ObjID	The *name* of the *namespace*.	RO	Y	Text. A name of "Default" is prohibited.
RecipeReadOnly-Level	The level of *approval* at which recipes are *read-only*.	RW	Y	Unsigned integer.
Members	The *names* of *agents* capable of *verifying* and *executing* the recipes in the *namespace*.	RW	Y	List of *agent names* (identifiers).
SynchOn	Level of synchronization (see Section 9.5). Required if synchronization is supported.	RW	N	Unsigned integer: Either 0 = Disabled, or any combination (sum) of: 1 = changes in body 2 = new *execution recipe* 8 = changes in *last value* 16 = new *derived object form execution recipe*
SynchFail	Specifiers for *execution recipes* for which an attempt to *synchronize* failed. Required if *synchronization* is supported.	RW	N	List of formatted text.

Table 4.2 defines the attributes of the *recipe namespace manager* objects. The *name (identifier)* of the *manager* is not generally of interest, as the *namespace specifier* is more commonly used. However, it is important for the *manager* to be accessible through Object Services.

Table 4.2 Recipe Namespace Manager Attribute Definition

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RNS_Mgr"
ObjID	The *manager's* name.	RO	Y	Text.
NamespaceName	The *name* of the *namespace* managed.	RO	Y	Text.

5 Distributed Recipe Namespace

A **distributed recipe namespace** is a *recipe namespace* that utilizes the storage capacity of multiple *agents* for recipe storage. Recipes are stored in special **recipe namespace segments** provided by the different *agents*. These *segments* are **distributed recipe namespace segments** and are supervised by the **distributed recipe namespace manager**.

This section defines the different objects that together provide the *distributed recipe namespace* capability: the *distributed recipe namespace*, the *distributed recipe namespace manager,* the *distributed recipe namespace segment*, and the **distributed recipe namespace recorder**. Detailed descriptions of operations are contained in Section 10.

Throughout this document, the acronym **DRNS** refers to the term "distributed recipe namespace" and is used primarily to differentiate a *DRNS* subtype object from its supertype.

5.1 *Motivations* — The *distributed recipe namespace* capability provides a method for using the storage capacity of multiple *agents*, as illustrated in Figure 5.1. This reduces the storage requirements of a centralized factory system, improves performance by allowing recipes to be used by those *agents*, and provides a centralized management of the *namespace* to ensure that the uniqueness of the recipe *identifier* is properly maintained.

Figure 5.1 illustrates a *distributed recipe namespace* with four *segments*, each provided by a different *agent*, and each *agent* also having a local *namespace*.

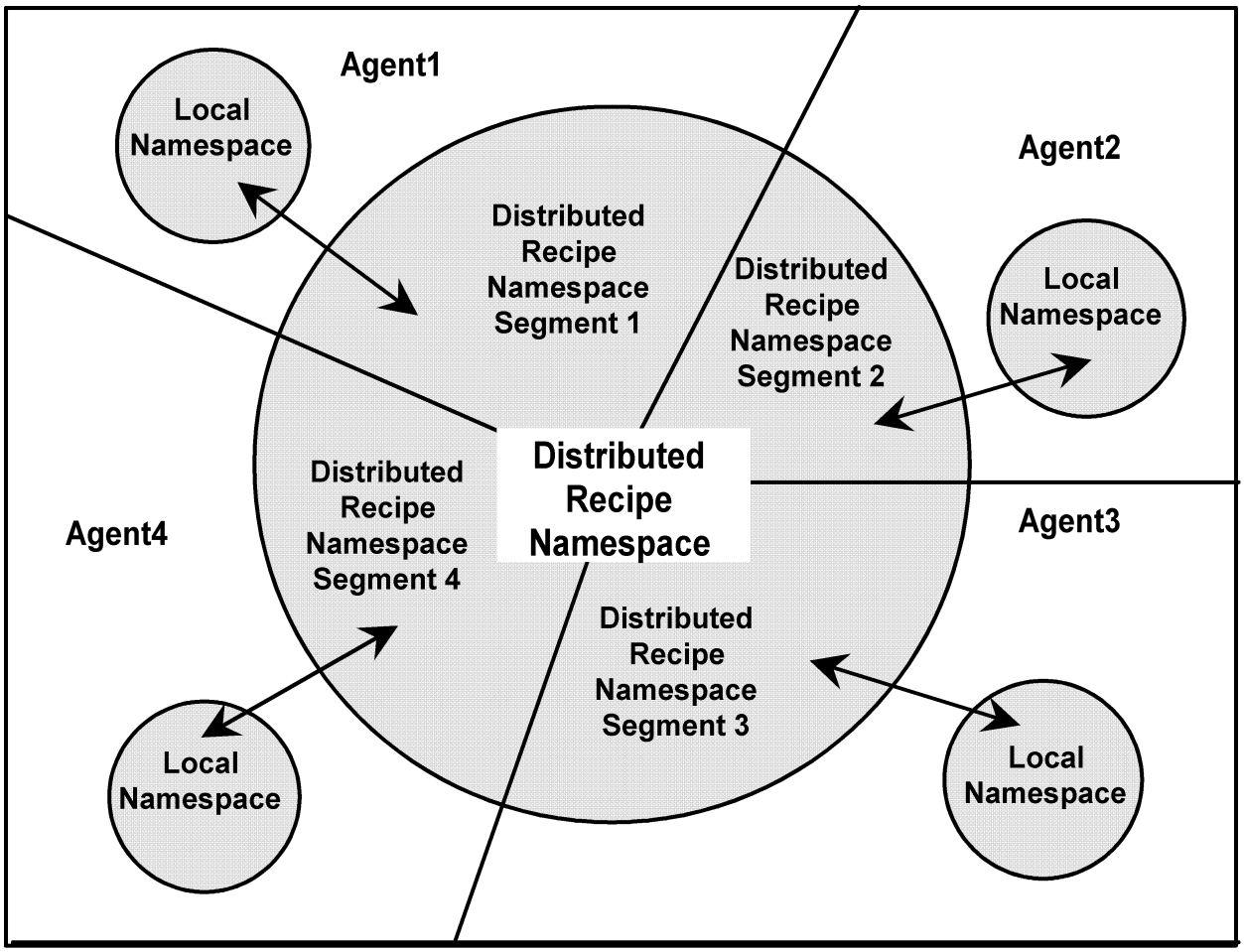


Figure 5.1
Illustration of Agents, Segments, and Local Namespaces

5.2 *Overview* — The *DRNS* object model (Figure 5.2) is a specialization of the model introduced in Section 4. The *distributed recipe namespace*, the *DRNS* segment, and the *DRNS* manager are subtypes of the *recipe namespace*, the *recipe namespace segment*, and the *recipe namespace manager* respectively. Each inherits the *attributes* and *operations* of its corresponding *supertype*. Only those attributes and operations that are specific to the *DRNS* types are shown in object representations.

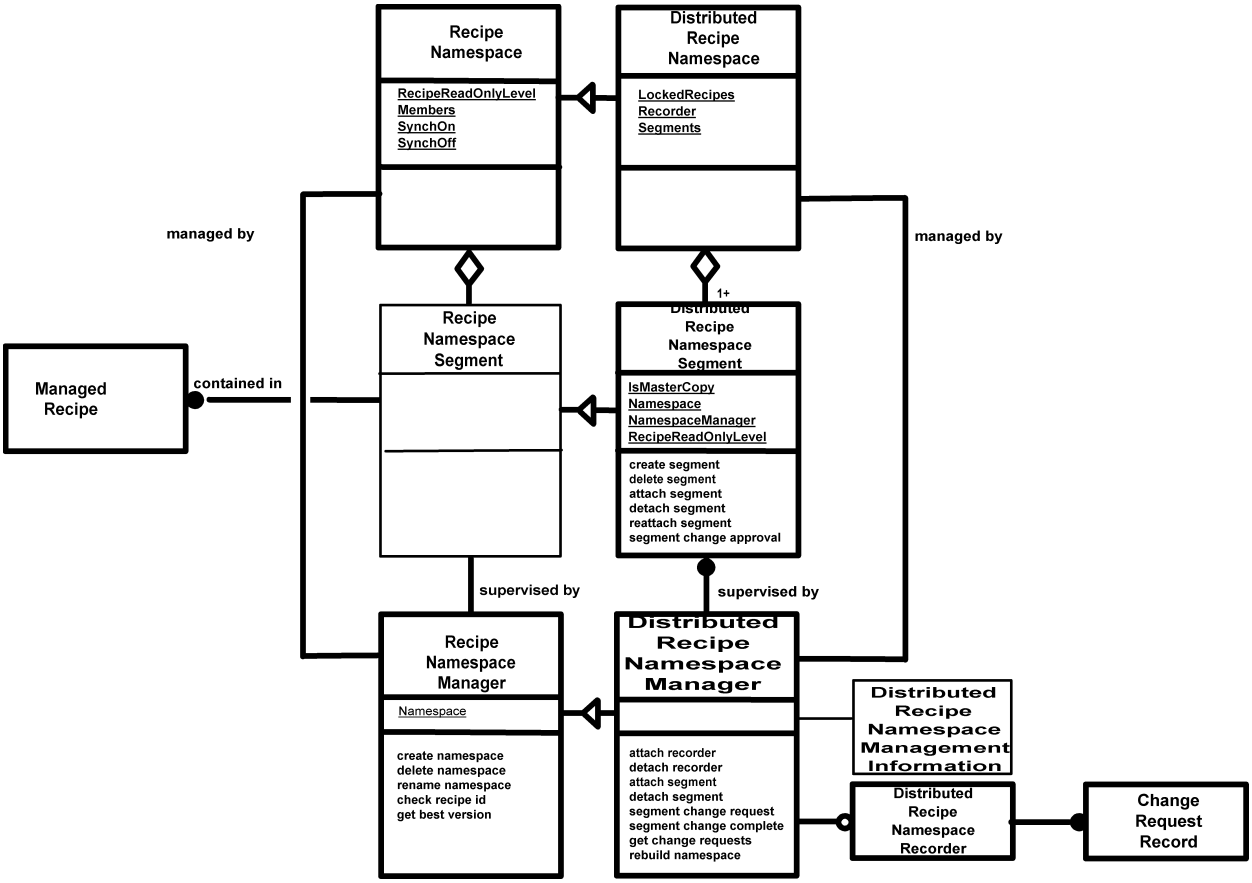


Figure 5.2
Distributed Recipe Namespace Model

The *namespace* supertype has exactly one *segment* of storage. Where storage is *centralized*, the *segment* is internal and private to the *namespace*. However, in the case of the *distributed recipe namespace*, there may be multiple *DRNS segments*, each provided by a separate *agent*. For this reason, the *DRNS segment* is a *standardized object*.

The *distributed recipe namespace* separates the management of recipes within a specific recipe storage area from the management of the entire *namespace*. The management of individual recipes within a specific storage area is delegated to the *distributed recipe namespace segment*. The *distributed recipe namespace manager* manages the various *segments* and the *namespace* itself. The *manager* is responsible for knowing the *identifiers* of all recipes stored in the entire *namespace*. It requires a knowledge of the structure of the recipe's *identifier* and the significance of the *version* in differentiating between recipes of the same name and different *versions*.

The *DRNS recorder* provides a backup facility for information required to automate the rebuilding of a *distributed recipe namespace*. It is external to the *DRNS namespace* and to its *DRNS manager*.

5.3 *Distributed Recipe Namespace Issues* — This section introduces issues that are applied to more than one object within the *distributed recipe namespace* capability.

5.3.1 *Object Services* — Attachment is a relationship between a managed object and its manager that is defined in OSS (SEMI E39). One object may be attached, detached, and reattached to a manager. When attached, the managed object is able to recognize that certain critical services have been requested by its manager. The *segment* and *recorder* objects shall comply with requirements for the operations and services defined in SEMI E39 (OSS) to attach, detach, and reattach to and from a *DRNS manager*. They shall also allow their *manager* to modify specified attributes that are otherwise read-only.

The ability to create and delete a *segment* or *recorder* is optional.

The *authorized user* may request a *DRNS manage*r to attach or detach a specified *segment* or *recorder*.

5.3.2 *Logical Recipe* — A **logical recipe** is defined as a recipe with a specific *body* and a specific set of *generic attribute values*. Every *managed* recipe stored is an instance of a *logical recipe*.

In a *distributed recipe namespace*, it is normal for instances (copies) of a given *logical recipe* to exist in more than one *DRNS segment* at any time. In other words, multiple duplicate copies of a recipe with the same *identifier* may co-exist within the *distributed recipe namespace*. To retain the integrity of the *namespace*, this can be allowed if, and only if, each copy of a recipe with a given *identifier* is an instance of the same *logical recipe*. It is the responsibility of the *DRNS manager* to ensure this logical identity is maintained.

Agent-specific datasets are not included in the definition of the *logical recipe* because multiple *agent-specific datasets* may exist independently of one another, and only the *dataset* specific to the *agent* providing the *DRNS segment* is normally kept in that *segment*.

5.3.3 *Change Requests* — A **change request** occurs whenever a *user*, an external application, or an attached *DRNS segment* requests the *DRNS manager* to make, or permit, any change in a recipe. Information concerning *change requests* is kept in the form of logical **change request records**. Once a *change request* is made, a *change request record* is created and maintained until the change has been either completed or discarded. Because the information represented by *change request records* is publicly available, they provide a degree of diagnostic capability.

The *change request record* is not a formal or standardized object, but it represents the information that is available through services.

Change management is the most critical issue of the *DRNS* capability and is discussed in detail in different sections below.

5.4 *Distributed Recipe Namespace Segment* — The **distributed recipe namespace segment** (Figure 5.3) is responsible for all of the activities that directly manipulate recipes, including storage, retrieval, deletion, and operations that change a recipe's *attributes* or *body*. This requires a micro-level knowledge of the recipe's *identifier*, its structure, the inter-relationships between the various recipe *attributes*, how the recipe is stored, and how it is transferred. It also includes a macro-level knowledge of all of the *identifiers* of the recipes that it has stored. The *DRNS segment* shall ensure that only one recipe with a given *identifier* exists within that *DRNS segment*. The contents of an unattached *DRNS segment* shall be *read-only*.

A *DRNS segment* and its storage are provided by an *agent*, which could be equipment, an independent "recipe server", or other factory systems.

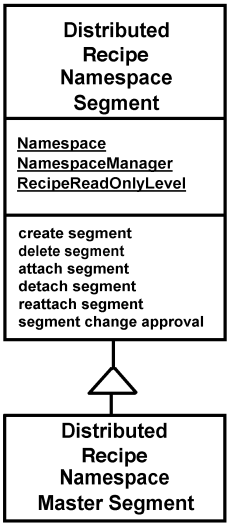


Figure 5.3

Distributed Recipe Namespace Segment

5.4.1 *Master and Dedicated Segments* — A **master segment** is a specialization of a *DRNS segment* that is capable of storing multiple *agent-specific datasets* per recipe. *Master segments* are not dedicated to a single equipment and are used to store a **full copy** (including all existing *agent-specific datasets*) of all *logical* recipes within the *namespace*. It is required that every

distributed recipe namespace manager be capable of supporting at least one attached *master segment*. Additional *master segments* may be desirable for further backup protection.

The term **dedicated segment** is used to refer to *segments* that are not *master segments*. *Dedicated segments* store at most one *agent-specific dataset* per recipe. Because a *DRNS segment* provided by equipment would not expect to keep *agent-specific datasets* for other *equipment*, equipment normally does not provide *master segment* capability.

Master and *dedicated* segments have the same attributes and support the same message services, but they respond differently to *agent-specific datasets* and are used for different purposes. A *master segment* requires approval prior to changing an *agent-specific dataset*.

5.4.2 *Change Restrictions* — The *DRNS segment* may provide access to recipes within its storage to the other *components* of the *agent* providing the storage. However, neither the generic attributes nor the body of recipes stored by the *segment* shall be changed except with the explicit approval of the *distributed recipe namespace manager*.

Dedicated DRNS segments that are attached to a *DRNS manager* may change the contents of an *agent-specific dataset* without first asking permission. However, they are required to notify the *manager* of any change as soon as it occurs.

The attribute RecipeReadOnlyLevel is set by the *DRNS manager* to the value of the corresponding *namespace* RecipeReadOnlyLevel attribute when the *segment* is first attached and whenever the *namespace* attribute is changed. This attribute has the same function as the *namespace* attribute and allows the *segment* to prohibit changes based on a recipe's *approval level* as defined in Section 8.2.5. For example, the *segment* shall deny requests to *modify a write-protected* recipe.

The *DRNS segment* is prohibited from changing its contents whenever it is unattached. However, recipes and recipe *attributes* may be read at any time by other entities, including other *components* of the owner *agent*.

Any changes to a logical recipe stored by the *DRNS segment* shall first be approved by the *DRNS manager* before the change is made. This includes any changes to the body or to any generic attribute. This is required for two reasons. First, two different *DRNS segments* may attempt to change a recipe at the same time, and this activity must be coordinated. Second, to protect the integrity of the recipe *identifier* where multiple instances of a recipe exist, the *distributed recipe namespace* is required to ensure that all such instances have been updated appropriately with that change before other changes to the same recipe are allowed.

NOTE: The prohibition against unauthorized change does not preclude the saving of such changes external to the *DRNS segment* while waiting for authorization. However, changes made by other *DRNS segments*, subsequent to such changes and prior to authorization, may invalidate these changes.

Communications between the different *components* of an *agent* that do not require or use the formal *services* defined by RMS are considered as proprietary to the *agent* and are neither covered nor excluded by RMS, subject to the above restrictions against change.

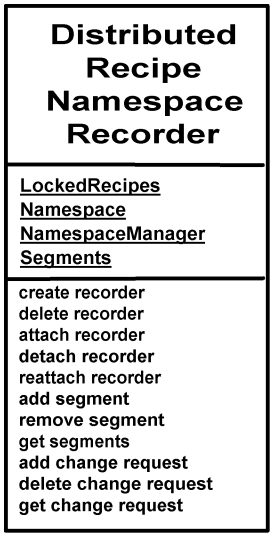


Figure 5.4

Distributed Recipe Namespace Recorder

5.5 *Distributed Recipe Namespace Recorder* — The **distributed recipe namespace recorder** provides a method of externally storing and retrieving information critical to automated rebuilding of a damaged *distributed recipe namespace*. It contains two types of information: a list of the *DRNS segments* that are attached to the *DRNS manager* and the current *change request record* in process per recipe.

A *DRNS recorder* may be attached and detached from a *DRNS manager*, and reattached to that *manager*. Information that it contains is available to anyone but may only be changed by its *manager*.

The *DRNS recorder* is able to store the *DRNS* **segment specifiers** (the *object specifiers*) of the attached *DRNS segments* in its Segments attribute. This information allows a *distributed recipe namespace* to be automatically rebuilt in the event that the *namespace*, or

the *DRNS manager*, is damaged and its information becomes lost or unavailable.

The *DRNS recorder* is also used to store the current *change request* per recipe, to delete a *change request*, and to return the set of *change requests* for one or more recipes. The attribute LockedRecipes provides a list of *identifiers* for recipes with *change requests*.

The *DRNS manager* shall use the services of a *DRNS recorder* when one is attached to the *namespace* by a service user. However, use (attachment) of a *recorder* is optional for the user. The *DRNS recorder* is provided for remote storage of critical information and is not intended as a general source of information for the user. It is not able to provide *inactive change request* information.

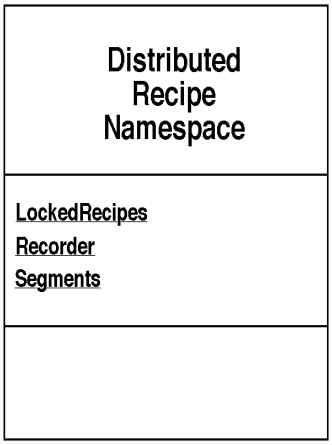


Figure 5.5

Distributed Recipe Namespace

5.6 *Distributed Recipe Namespace Management Information* — The *DRNS* **management information** object is private and proprietary to the *DRNS manager*. It is included in Figure 5.2 as an emphasis on the importance of the information that a *DRNS manager* requires for management. This includes, but is not limited to, the recipe *identifiers* stored in each attached *segment* and all existing *change request records*.

The *DRNS manager* is required to know the contents of all of its attached *segments* at all times. It shall be able to uniquely identify each instance of a recipe within the *distributed recipe namespace*. It is responsible for tracking the current status of each instance of a logical recipe within the *distributed recipe namespace* when a change to a recipe within one *segment* is being updated to other *segments*. The *DRNS management information* is important for these purposes.

5.7 *Distributed Recipe Namespace* — The *distributed recipe namespace* has three additional attributes, as shown in Figure 5.5, that are read-only, set by the *DRNS manager*. The attribute Segments contains a list of the *object specifiers* of the *DRNS segments* currently attached. The attribute Recorder contains the object specifier of an assigned *DRNS recorder*. The attribute LockedRecipes contains a list of recipe *identifiers* of recipes with existing *change request records*.

5.8 *Distributed Recipe Namespace Manager* — The *distributed recipe namespace manager* (Figure 5.6) is responsible for ensuring that the *distributed recipe namespace* and *DRNS segments* operate together and for maintaining *namespace* integrity as a whole. It is required to know the identities of its attached *DRNS recorder* and *DRNS segments* and the contents (*recipe identifiers*) of each attached *DRNS segment* at all times.

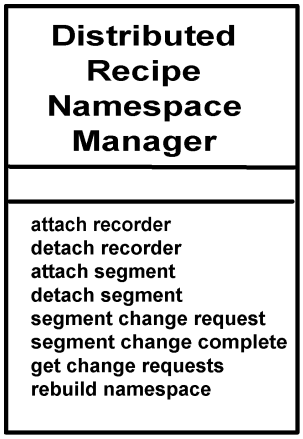


Figure 5.6

Distributed Recipe Namespace Manager

The current attachments of the *DRNS manager* are maintained in the *DRNS namespace* attributes Recorder and Segments and through the services of the *DRNS recorder* described in Section 5.5. Whenever a *DRNS segment* is attached, the *DRNS manager* shall add the *segment specifier* to the *namespace* attribute Segments and to the current list maintained by the *DRNS recorder*. Whenever a *DRNS segment* is detached, the *DRNS manager* shall remove its *specifier* from the Segments attribute and from the *DRNS recorder's* list.

The *DRNS manager* is able to know which recipes are stored in each *DRNS segment* through use of Object Services provided for or by the individual *DRNS segments*. It shall maintain the integrity of the *namespace* through ensuring that an *identifier* of any recipe stored within the *namespace* represents exactly one *logical recipe*. This is achievable because the *manager* must give explicit approval of any change of *logical recipes* (generic attributes or body) stored by the *DRNS segments*.

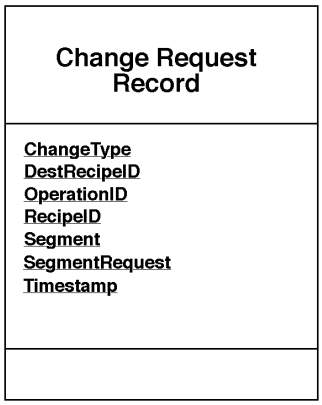


Figure 5.7

Change Request Record

5.8.1 *Change Management* — A *change request* occurs whenever a *user* or an attached *segment* requests the *distributed recipe namespace manager* to make, or permit, a specific type of change. A change request record (Figure 5.7) contains information about who requested what type of change. This allows later diagnostics when necessary.

A *change request record* is created for a recipe whenever an attached *DRNS segment* requests a change and is deleted when the requested change has been completed or discarded. A *change request* is either **inactive** or **active**. Once a *change request* is made, it is considered as inactive until the change has been approved by the *DRNS manager*. Once approved, it is considered as **active**.

The *change request record* is not a standardized object and is not accessed directly through public services. However, the information represented by the *change request record* shall be available, upon request, from the *DRNS manager* for all *inactive* and *active change requests*. For this reason, it is convenient to model the *change request record* as an object.

A recipe for which a *change request* exists is called **locked**. Otherwise, it is **unlocked**.

The *DRNS manager* is responsible for ensuring that only one *change request* per recipe is *active* at any time. The results of each approved *change request* shall be updated appropriately to each of the other *DRNS segments* with a copy of the same *logical recipe* before a subsequent change to the recipe is approved.

5.9 *Building a Distributed Recipe Namespace* — A *distributed recipe namespace* is built up in several stages. The *distributed recipe namespace* and its associated *DRNS manager* are created separately in the first stage. The individual *DRNS segments* are first *created* and then attached to the *DRNS manager*. At least one *DRNS segment* must be attached to the *DRNS manager* before it can accept recipes, as the *distributed recipe namespace* has no other means of storage. Figure 5.8 illustrates the attachment relationship.

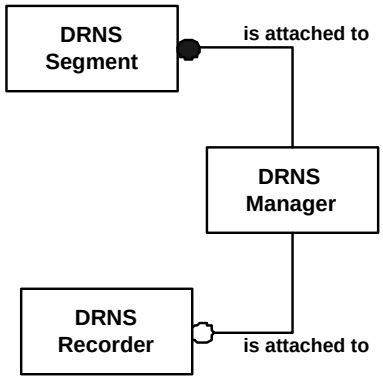


Figure 5.8

The Attachment Relationship

The attached *DRNS segments* may later be detached from the *namespace* and then either attached again to the same *namespace* or to a different *namespace* without affecting the recipes stored within the *DRNS segment*.

NOTE: When a *DRNS segment* with existing recipes is detached and then attached to a different *namespace*, its recipe *identifiers* fall within the domain of the new *namespace*. This may require some recipes to be *renamed* before the attach process is complete.

5.10 *Rebuilding a Damaged Distributed Recipe Namespace* — A *distributed recipe namespace*, or its *manager*, may become damaged or unavailable. If a *DRNS recorder* was attached to the *damaged* name-space, then rebuilding the *namespace* can be automated.

A new *distributed recipe namespace* and new *DRNS manager* are created, assigning the old *namespace* name as the object identifier ObjID for the new *namespace*. The new *DRNS manager* should be assigned a different *identifier*, however, as a security measure. The *user* may then request the new *DRNS manager* to rebuild the *namespace* with the old *DRNS recorder*. A *namespace* may also be rebuilt without a *recorder* if the list of *segment specifiers* can be provided by the user. However, the user is not expected to provide the information retained through *change request records*.

5.11 *Object Attribute Definition Tables* — This section contains the formal attribute definitions for the *distributed recipe namespace* capability. Except for the *object identifier* attributes, attributes are listed alphabetically.

Objects that are subtypes of objects introduced in Section 4 inherit the attributes of the supertype objects. These attributes are not repeated in this section.

5.11.1 *Distributed Recipe Namespace Segment Attribute Definition* — Table 5.1 defines the attributes required for a *distributed recipe namespace segment*, and Table 5.2 defines the attributes of the subtype *master segment*.

Table 5.1 Distributed Recipe Namespace Segment Attribute Definition

Attribute Name	*Description*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RNSDSegment"
ObjID	The object name (identifier).	RO	Y	Text.
Namespace	The name (ObjID) of the *namespace* to which the *segment* belongs. May be set by the manager.	RO	Y	Text.
NamespaceManager	Identifies the *distributed recipe namespace manager*. May be set by the manager.	RO	Y	Text.
RecipeReadOnlyLevel	Used to track the corresponding attribute of the *namespace* to which the *segment* belongs. May be set by the manager.	RO	Y	Unsigned integer.

Table 5.2 Distributed Recipe Namespace Master Segment Attribute Definition

Attribute Name	*Description*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RNSDMaster"
ObjID	The object name (identifier).	RO	Y	Text.

5.11.2 *Distributed Recipe Namespace Recorder Attribute Definition* — Table 5.3 defines the attributes required for a *distributed recipe namespace recorder*.

Table 5.3 Distributed Recipe Namespace Recorder Attribute Definition

Attribute Name	*Description*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RNSDRecorder"
ObjID	Text.	RO	Y	Text.
LockedRecipes	List of *identifiers* of recipes with existing *change request records*.	RO	Y	List of formatted text.
Namespace	Identifies the *namespace* to which the recorder is attached. May be set by the manager.	RO	Y	Text.
NamespaceManager	Identifies the *distributed recipe namespace manager*. May be set by the manager.	RO	Y	Text.
Segments	List of *specifiers* of currently attached *segments*.	RO	Y	List of formatted text.

5.11.3 *Distributed Recipe Namespace Attribute Definition* — Table 5.4 defines the attributes required for a distributed recipe namespace.

Table 5.4 Distributed Recipe Namespace Attribute Definition

Attribute Name	*Description*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RNSD"
ObjID	Text.	RO	Y	Text.
LockedRecipes	A list of *identifiers* of all recipes with existing *change request records*.	RO	Y	List of formatted text.
Recorder	The *recorder specifier* of the attached *distributed recipe namespace recorder*.	RO	Y	Text.
Segments	A list of *specifiers* of the *distributed namespace segments* attached to the *namespace*.	RO	Y	List of formatted text.

5.11.4 *Distributed Recipe Namespace Manager Attribute Definition* — Table 5.5 defines the attributes required for a *distributed recipe namespace manager*.

Table 5.5 Distributed Recipe Namespace Manager Attribute Definition

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RNS_MgrD"
ObjID	The *manager's* name.	RO	Y	Text.

6 Recipe Executor

This section describes the basic concepts for the *recipe executor* and the *execution recipe* that it stores.

The **recipe executor** is the component of an *executing agent* that reads and comprehends the contents of a recipe (its *body*) and puts into effect its instructions, settings, and/or other data. The object model of the *recipe executor* is shown in Figure 6.1.

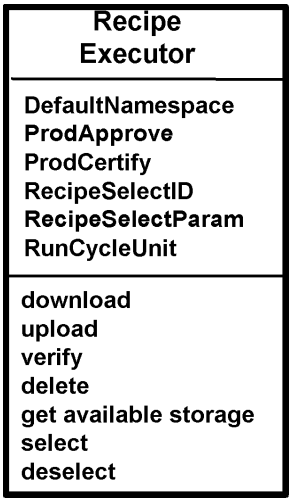


Figure 6.1

Recipe Executor

The *recipe executor* is able to temporarily store recipes for execution purposes, and it may also be able to store *execution recipes* for later execution.

The execution process is beyond the scope of RMS.

6.1 *Motivations* — Specification of the *recipe executor* and the *execution recipe* that it stores is necessary to complete the management of recipes in the factory. The *recipe executor* provides limited storage for recipes and minimum capability to manage them. The storage provided is intended to be temporary only.

Some *recipe executors* have the ability to purposefully change a recipe's body or create new recipes. To ensure that *execution* recipes remain synchronized with the *managed* recipes in a *recipe namespace*, additional rules are required for such cases.

The requirements for RMS that are addressed by the *recipe executor* include the following:

- *The ability to create, edit, and change recipes outside the* executing agents *that execute them,*
- *The ability for* executing agents *to use recipes developed externally,*
- *The ability to share recipes among equipment of the same type,*
- *Protection of stored* execution recipes *and of currently* selected *recipes from unexpected or unauthorized change,*

- *Execution of recipes without errors, including errors detected during* verification *and* ***validation*** *errors caused by improper settings or parameters or incompatibility with the current configuration,*
- *The ability to change a recipe's* parameters *between runs in a systematic way without changing the recipe itself,*
- *The ability to dynamically connect and disconnect the* executing agent *from the communications environment, and*
- *The ability of standalone equipment to execute recipes without a communications link.*

6.2 *Description* — The *recipe executor* is able to receive a downloaded *execution recipe* and temporarily store it. The *executor* can *verify* the recipe *body* (Section 11.2.2) both at the time of the download and after the recipe has been stored. It is able to store at least as many recipes as it requires for a single process cycle, which is determined by its own requirements.

Recipes stored by the *recipe executor* may be downloaded from more than one *namespace*. To prevent ambiguity or conflict between recipe *identifiers* from different *namespaces*, the name of the namespace from which the recipe was originally downloaded is retained in the *identifier* of the *execution recipe*.

The *recipe executor* **selects** one or more specified recipes by *validating* them and preparing them for execution. This may include moving the recipes into a separate **recipe execution area** to create an **executable copy** recipe. The *executable copy* recipe, if it exists separately, shall be protected from inadvertent change caused by other activities, such as downloading a new recipe. The *executable copy* recipe is not otherwise addressed by RMS. If a separate *copy* is *selected*, the stored *execution recipe* shall be protected from change. Protection from change and permission to change are discussed in Section 6.6.

Validation of a recipe consists of checking the values of its settings and variable parameters against existing supplier-defined and/or user-defined restrictions, and ensuring the recipe, or the *linked recipe set*, is valid for the current configuration of the *executor* (e.g. equipment or attached module).

Additionally, the *recipe executor* is able to calculate the amount of available storage, to delete recipes from its storage to make room for new recipes, to **de-select** recipes by preventing them from being re-executed without another explicit *select*, to **rename** an *execution recipe*, and to provide requested information about itself and its stored *execution recipes*, in conformance with OSS.[8]

The *recipe executor* may wish to rewrite *source form* recipes into a proprietary *derived object form* that is more efficient for execution or storage purposes. Where this type of recipe is to be stored for re-use, a new *identifier* is required for the *object form*, as described in Section 3.2.2.1.2. *Derived object form* recipes are discussed in detail in Section 11.2.2.1.

6.3 *The Execution Recipe* — An *execution recipe* is a type of recipe, as shown in Section 3, Figure 3.5. The *recipe executor* stores recipes as *execution* recipes. *Execution* recipes are created in one of two ways: they are either **downloaded** from a *recipe namespace*, or they are **created** by the *recipe executor*.

6.3.1 *Comparison of Managed and Execution Recipes* — An *execution recipe* differs from a *managed recipe* in two ways: its attributes and its lack of *agent-specific datasets*.

The differences in attributes between a *managed recipe* and an *execution recipe* consist of:

- *The addition of* Namespace *as an* identification *attribute of the* execution recipe*,*
- *The intermediate parameters,* ExtRef *and* Parameters*, of a* verified managed *recipe, required for the* namespace link *operation, are not used by the* recipe executor *and are not retained in the* execution recipe*,*
- *The attributes* LinkParam *and* AgentSpec LinkParam *of the* managed recipe *are merged into the single attribute* ExecLinkParam*,*
- *The attributes of the remaining* agent-specific dataset *are absorbed into the attributes of the* execution *recipe, and*
- *The addition of the attribute* length ExecAttrLength *and attribute* timestamp ExecAttrChgTime *attributes, and*
- *The additional attribute* ChangedBody*, which is required for* execution recipes *where the* recipe executor *is capable of changing the recipe* body *or of creating new recipes and is not otherwise used.*

The conversion of a *managed recipe* to and from an *execution recipe* is the responsibility of the *namespace manager* and is discussed in Sections 9.4.8 and 9.4.9.

Most of these differences are invisible for an *unverified* recipe or minimal recipe implementations. For an

8 SEMI E39 (Object Services Standard)

unverified recipe, all of the above attributes are *cleared* to their *default value* and are not transferred with the recipe. Attributes needed for multi-part recipes and *variable parameters* are not required for minimal implementations. Figure 6.2 provides a model of the minimal implementation of an *execution recipe* to meet RMS requirements.

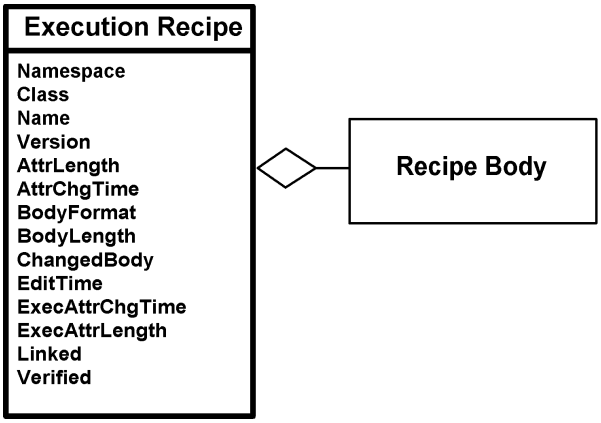


Figure 6.2

Object Model for Minimal *Execution* Recipe

6.3.2 *Downloaded Recipes* — To *download* a recipe from a *namespace*, the *namespace manager* is responsible for first converting the attributes of the specified *managed recipe* into those of the *execution recipe*. The *recipe executor* stores the downloaded recipe as an *execution* recipe. The body of the recipe is unchanged by the download operation and may be in *source form* or *object form*, including the *derived object form*. (See Section 11.2.2.1.)

The *namespace manager* may optionally request that the recipe being downloaded replace any pre-existing recipe with the same *identifier*. Otherwise, if such a recipe exists, the download request is denied.

Upon request, the *recipe executor* shall *verify* a recipe that it has previously stored and return to the requestor the information required for recipe management.

Attributes of the *execution* recipe that have the same attribute *name* as either a *generic* attribute or an *agent-specific* attribute of the *managed recipe* are not changed except in specific cases.

The attributes giving the *timestamp* and *length* of the attributes of the *execution recipe* are different from those of the *managed recipe*. This allows the *generic* and *body descriptors* of a *managed recipe* to be compared with the corresponding attributes of an *execution recipe* for traceability. The *timestamp* of the *execution recipe's* attributes is ExecChgTime, and the *length* of its attributes is ExecAttrLength. These are calculated by the *recipe executor* at the time the *execution recipe* is stored and updated whenever other attributes of the recipe change.

6.3.3 *Execution Recipe Identifier* — The *identifier* of an *execution recipe*, in addition to the recipe *name, class,* and *version* of the *managed* recipe, also contains the name of the **originating namespace**. The **originating namespace** is the namespace from which the recipe was originally downloaded. For recipes newly created within the storage of the *recipe executor*, this is the namespace to which the recipe will be *uploaded*.

The full *identifier* of an *execution recipe* is identical to that used within recipes to indicate the recipe is external to that of the referencing recipe (see Section 3.2.4.1.1) and has the form:

"Namespace Name>/CLASS/.../CLASS/name; version".

6.3.4 *Execution Recipe Descriptor* — The *descriptors* of the *execution recipe* consist of the *execution attribute descriptor* (ExecAttrLength and Exec-AttrChgTime), the *generic attribute descriptor* (AttrLength and AttrChgTime), and the *body descriptor* (BodyLength and EditTime). The **execution recipe descriptor** consists of the *execution attribute descriptor*, the *generic attribute descriptor*, and the *body descriptor*, in that order.

6.3.5 *Execution Recipe Attribute Definitions* — Table 6.1 provides the formal definition of the attributes of the *execution recipe*. Attributes in Table 6.1 are classified as identification attributes, mandatory and other required attributes, optional attributes, and non-standard attributes. Identification attributes are not used in RMS services as "recipe attributes" and are used solely to specify one or more execution recipes. Support for all required attributes is necessary for RMS compliance. Support for the remaining attributes is not required.

The *execution* recipe's attribute *length* and *timestamp* attributes are transferred first when a recipe **attribute section** (Section 14.1) is **uploaded**.

Attributes that are described as "preserved" are maintained without change from a *downloaded* recipe. For recipes that are first created by the *recipe executor*, such as hardware-specific recipes, they are set to their appropriate values. Default values have been added for attributes that cannot be determined.

Attributes in Table 6.1 are presented in the following order:

- *Identification attributes, including object type and identifier, the first six listed in the table,*

- *Mandatory attributes and the other required attributes, in the order in which they are to appear when a recipe is transferred.*
- *Optional attributes in alphabetical order, and*
- *Non-standard attributes, transferred last.*

Default values for *non-mandatory* attributes are shown in the last column. Null values (indicated by "NULL") are dependent upon the particular form (see Section 3.2.5).

Table 6.1 Execution Recipe Attribute Definition

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*	*Default Value*
Identification Attributes					
ObjType	The object type.	RO	Y	Text: "ERcp"	"ERcp"
ObjID	An identifier derived from Namespace, Class, Name, and Version	RO	Y	Formatted text.	-
Namespace	The name of the *originating namespace.*	RO	Y	Text.	NULL
Name	A logical name assigned by the user when the recipe is *created*.	RO	Y	Text.	-
Class	The recipe's class (e.g., "/PROCESS/" or "/PROCESS/LOADER/").	RO	Y	Formatted text: "CLASS/CLASS/../ CLASS/"	-
Version	The version of the recipe.	RO	Y	Text.	-
Mandatory Attributes					
ExecAttrLength	The *length attribute* for the attributes of the *execution recipe*. Calculated when the recipe is *downloaded* and whenever an attribute changes.	RO	Y	List of formatted text.	-
ExecChgTime	The *timestamp* of a change to the attributes of the *execution recipe*.	RO	Y	Formatted text, *timestamp* format.	-
AttrLength	Preserved.	RO	Y	Unsigned integer.	0
AttrChgTime	Preserved.	RO	Y	Formatted text.	NULL
Required Attributes					
BodyLength	Preserved unless recipe is modified. Length of the recipe's body, in bytes.	RO	Y	Unsigned integer.	-
EditTime	Preserved unless recipe is modified. *Timestamp* of when the *body* was *created* or modified.	RO	Y	Formatted text. *Timestamp* format.	-
BodyFormat	Indicates the form and format of the recipe's *body.*	RO	Y	Enumerated unsigned integer: 0 = *source*, 1 = *object*, > 1 reserved.	0
Verified	Indicates whether the recipe's body is syntactically correct.	RO	Y	Boolean.	FALSE
Linked	Indicates whether the recipe is *linked*.	RO	Y	Boolean.	FALSE
ChangedBody	Set to TRUE if the recipe body has changed without a subsequent upload to the originating namespace. NOTE: This attribute is never uploaded to a namespace. Required only if recipe can be changed or created.	RO	Y	Boolean.	FALSE

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*	*Default Value*
ExecChgCtrl	Preserved. Specifies change control requirements for recipe.	RO	Y	Binary. Bitwise (MSB=8): 1 - The recipe body may be changed, 2 – Change notification required, 4 - Recipe may be selected after change, 8 - Most recent parameter setting shall be saved.	0
Optional Attributes					
AgentSpecComments	Copied from the original *agent-specific* attribute when downloaded. Set by the user.	RO	N	Text. Maximum length is 80 characters.	-
ApprovalLevel	Indicates the level of approval assigned by an *authorized user.*	RW	N	Unsigned integer.	0
Certified	Preserved from the *agent-specific* attribute as downloaded. May be used as control for production-worthy recipes.	RO	N	Unsigned integer.	0
Comments	User comments. Preserved from the *generic* attribute as downloaded.	RO	N	Text. Maximum length is 80 characters.	-
EditedBy	Preserved unless recipe is modified. The name of the person or *executing agent* who last modified the recipe.	RO	N	Text. Maximum length is 80 characters.	-
EstRunTime	The nominal or estimated execution (run) time of the recipe, in seconds. Used for scheduling purposes. Preserved from the *generic* attribute as downloaded.	RO	N	Unsigned integer.	0
ExecLinkParam	Preserved unless *last value* is changed (Section 6.6.4). Contains the list of *parameter definitions*, including any *agent-specific* modifications. Required for *variable parameter* support.	RO	N	Structure composed of parameter name, initial value, and restrictions.	NULL
LinkList	Preserved. A complete list of recipe *specifiers* for a *linked recipe set*. Required for multipart recipe support.	RO	N	List of formatted text.	NULL
SrcRcpID	For a *derived object form* recipe, contains the recipe *identifier* of the original *source form* recipe. Required only for *derived object form* recipes.	RO	N	Formatted text.	NULL
VerificationID	Identifier code used by the *verifier* of the recipe. May be used to determine out-of-date formats that need to be *reverified*.	RO	N	Text. Maximum length is 40 characters.	NULL
Non-Standard Attributes					
AgentSpec- UD *	Preserved from the original *agent-specific* attributes as downloaded.	RO	N	Defined by supplier or user. Text limited to 80 characters.	-

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*	*Default Value*
UD *	Non-standard attribute defined by supplier or *user*. Asterisk indicates the part of the attribute name that is provided in this definition. Shall be preserved exactly, except by the entity that defined it.	RO	N	Varies with definition. Text form is limited to 80 characters.	-

6.4 *Default Namespace* — The **default namespace** is a *dedicated centralized namespace* (see Section 4.4) that is used for all *agent-specific* recipes. A *recipe executor* that uses *agent-specific* recipes, such as thermocouple calibration tables, shall be provided with a *namespace* for such recipes. *Equipment* that may be operated in a stand-alone mode, and that requires *agent-specific* recipes, shall also provide a local *namespace* to be used for this purpose. A single *namespace* shall be used to satisfy both requirements.

The *recipe executor* shall provide a user-settable *attribute* DefaultNamespace that contains the name of the *default namespace*. The *default namespace* shall be available on power-up for stand-alone operation. For *executing agents* intended to operate only in a *supervised* configuration, such as cluster process modules, the *default namespace* may be provided by the *supervisor*.

A recipe in a *default namespace* is referenced within a recipe *body* by specifying a *namespace* named "Default". This allows controlled specification by *namespace* role for hardware-specific recipes.

6.5 *Recipe Storage* — Discussion of types of storage used by the *recipe executor* is provided to clarify terminology.

Storage is generally assumed to consume space in some form, and the amount of space available for recipes is assumed to be finite, so that adding recipes reduces the amount of space available and deleting recipes increases the amount of space available. These assumptions are based on current technologies and are not requirements.

The *recipe executor* may have one or more types of storage area for recipes, shown in Figure 6.3. The storage area used for the current process cycle is called the **recipe execution area**. This is the minimum storage capacity required. *Executable copy* recipes in the *recipe execution area* may or may not be transformed for execution purposes but shall retain the attributes of the *execution recipe*. The *recipe execution area* shall be protected from all inadvertent and unintentional change, including change resulting from transferring a recipe to or from a *recipe namespace* or from *editing* a recipe.

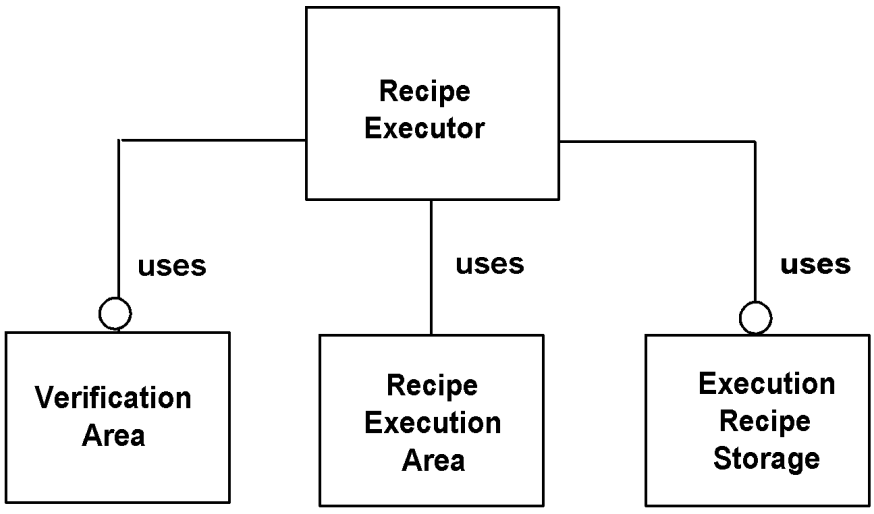


Figure 6.3

Object Model for Recipe Executor

Extra storage area, for additional recipes that are not currently *selected*, when provided, is called the **execution recipe storage**.

A separate intermediate area for temporary storage of *unverified* downloaded recipes may also be provided. This area is the **verification area**. The separation of this area provides protection for the *executable copy* recipes and the *execution recipes*. Recipes that fail

verification, or that have been requested to be discarded without storage after *verification*, may be removed from this area more easily when it is separate.

A recipe temporarily placed in a *verification area* shall be immediately either discarded or else moved to *execution recipe storage* following a successful *verification*.

The term **stored recipes** in this section refers to recipes in *execution recipe storage*.

The storage areas of the *recipe executor* may or may not be volatile. The *verification area*, *recipe execution area*, and the *execution recipe storage* are not required to be physically separate, so long as recipes in each logical area are protected from change caused by activities in the other areas.

6.6 *Change Control* — Storage provided by the *recipe executor* is intended as temporary, and only minimum management capability is required, such as the ability to rename or delete recipes.

Some *recipe executors* may choose to provide additional capabilities, such as the ability to *create* new recipes, to *edit* existing recipes, to build a compressed *derived object form* recipe, to modify existing recipes through the execution process itself, and/or to save the **last value** used for *variable parameter settings*. Certain restrictions apply to these activities. Each different capability provided for creating and changing recipes shall be explicitly documented by the supplier.

In general, change is controlled by the *user* through the attribute ExecChgCtrl. ExecChgCtrl specified four separate types of control related to change, including permission to subsequently *select* or *re-select* a changed recipe and a requirement that the *originating recipe* be notified of all protected changes to the recipe.

The *recipe executor* is prohibited from setting the Linked attribute of any recipe to TRUE.

The *recipe executor* is responsible for ensuring the uniqueness of the *identifiers* of the recipes that it stores.

Detailed requirements governing the creation of new recipes and the changes protected by ExecChgCtrl are defined in Section 11.

6.6.1 *Recipe Creation* — Certain *recipe executors* may be able to *create* recipes. This capability is allowed to cover the hardware-dependent recipes and the provision of editing services. Mechanisms for creating and changing recipes are beyond the scope of RMS. *Change notification* is required for all newly *created* recipes.

6.6.2 *Recipe Compression* — A *source form* recipe may be *compressed* to obtain a *derived object form* recipe, described in Section 3.2.2.1.2. This is not considered as a *newly created* recipe, as the *source form* and *derived object form* recipes achieve the identical process results. For this reason, most of the attributes of the *source form* recipe, including the Linked and ExecChgCtrl attributes, are passed to the *derived object form*. A *change notification* requirement for the *source form* recipe extends to the *derived object form* as well, including notification when the *derived object form* is built. Requirements for the *derived object form* recipe are defined in detail in Section 11.2.2.1.

6.6.3 *Changes to Stored Recipes* — The *recipe executors* may be able to **change** *an existing recipe by changing its body*. Certain hardware-dependent recipes may sometimes be *changed* by, or as a result of, the execution process itself. Recipes also may be *changed* through an **editing** activity, including interactive "teach" and automated "self-teach" operations provided by some systems. Except where expressly granted permission to change an existing recipe through the ExecChgCtrl attribute, the bodies of all recipes in storage shall be *protected from change*. This is not the same as the *write-protection* of the *namespace* in that *execution recipes* may be *deleted* and *renamed* by an *authorized user* and by the manager of the *originating namespace*.

Depending upon the value of the ExecChgCtrl attribute of the recipe, permission to **change** the *execution recipe* (stored in the *execution recipe storage*) is granted or denied in advance and allows case-by-case granularity. Unless explicitly granted permission, a *changed* recipe may not be subsequently *selected* or *re-selected*.

Purposeful change during the *execution* process for hardware-dependent recipes is included in the ExecChgCtrl attribute permission to change the recipe body. For example, furnaces may be able to update a "profile recipe" during a normal process cycle. Recipes changed purposefully by the execution process are assumed to represent the best, most up-to-date, and most valid version of a hardware-dependent *class* of recipe. Suppliers of *recipe executors* with this capability shall provide complete documentation of the *class* of recipe changed and the circumstances under which it is changed.

The ExecChgCtrl attribute of a recipe may require that the *originating namespace* be notified of change. **Change notification** consists of a *notification message* sent to the *originating namespace* that alerts the *namespace manager* that a recipe has been changed or *originated*. Where *change notification* is required through ExecChgCtrl, the *namespace manager* is responsible for subsequently uploading the recipe, assigning it a new *identifier* if necessary, and requesting the renaming of the *execution recipe* as necessary to remain *synchronized*.

6.6.4 *Last Value* — Some *recipe executors* may also want to save the **last value** set by the user for each *variable parameter* and use it as the new *initial value* of that *parameter* when the same recipe is rerun.

This is useful when internal conditions drift in a consistent manner over time, the parameters are occasionally modified to compensate for such drift, and the last *setting* used is, therefore, a better "default" than the one specified in the recipe itself.

Permission to save the *last value* may be expressly granted by the user in the recipe attribute ExecChgCtrl.

Care should be taken to prevent unintentional modifications.

6.7 *Production* — Equipment utilizations states are defined by SEMI E10 (Standard for Definition and Measurement of Equipment Reliability, Availability, and Maintainability (RAM)). These states include the PRODUCTIVE and STANDBY states used by the factory for normal production work. The attributes ProdApprove and ProdCertify are defined as required only for equipment supporting states defined in this document.

To ensure that only recipes authorized for production are executed while the *executing agent* is in the PRODUCTIVE state, or are *selected* while in the STANDBY state, the *authorized user* may set the values of the attributes ProdApprove and/or ProdCertify to non-zero values. Non-zero values in ProdApprove or ProdCertify represent minimums for a recipe's ApprovalLevel and Certified attributes, respectively.

The *recipe executor* is responsible for comparing the corresponding recipe attributes ApprovalLevel and Certified. When a recipe is *selected* for execution (implicitly or explicitly) while either of the PRODUCTIVE or STANDBY states is active, the value of ApprovalLevel is required to be equal to or greater than the value in ProdApprove, and the value in Certified is required to be equal to or greater than the value in ProdCertify. Otherwise, the *select* shall fail.

6.8 *Recipe Executor Attributes* — The *recipe executor* is owned by the *agent* that provides a *recipe execution resource*. The *recipe executor* in turn owns the recipes that it has stored. It shall support Object Services for its owned object types and for itself.

Table 6.2 defines the *attributes* of the *recipe executor* in alphabetical order.

Table 6.2 Recipe Executor Attribute Definition

Attribute Name	*Definition*	*Access*	*Rqmt*	*Form*
ObjType	The object type.	RO	Y	Text = "RcpExec"
ObjID	Text.	RO	Y	Text.
DefaultNamespace	The name of an *executing agent's namespace* used for all hardware-dependent and other *agent-specific* recipes.	RW	Y	Text.
ProdApprove	The minimum value of a recipe's *approval level* accepted during productive and standby states. Required for SEMI E10 support only.	RW	N	Unsigned integer.
ProdCertify	The minimum value of a recipe's *certification level* accepted during productive and standby states. Required for SEMI E10 support only.	RW	N	Unsigned integer.
RunCycleUnit	The process unit on which the calculation of the estimated value of the recipe *generic attribute* EstRunTime is based.	RO	N	Case-sensitive formatted text composed of a unit of measure and an optional numeric suffix. Compliant with SEMI E5, Section 9.
RecipeSelectID	A list of recipe *identifiers* for the currently selected recipes.	RO	Y	List of formatted text.
RecipeSelect-Parameters	A list of all *parameter definitions* in effect for the i^{th} recipe *identifier* in RecipeSelectID. The maximum value for i is determined by the equipment supplier as the maximum number of recipes which may be *selected* at the same time. Required if variable parameters are supported.	RO	N	List of structures composed of parameter name, parameter value, parameter restriction.

7 Agents

This section describes the *agent* and **resources** as they are used in Recipe Management. The concept of *agent* is introduced to cover the different types of RMS implementations and to provide a context for the other specific object types introduced in RMS.

7.1 *Definitions* — A **resource** is an owned entity that has an active role in factory operations. A factory has many different kinds of *resources*. Some *resources*, such as valves, may be primarily physical. A software application is a type of *resource* not generally considered as physical. The factory itself is a *resource* for the corporation.

An **agent** is a system in a factory — a type of *resource* that includes both hardware and software components, at least some of which are also *resources*. Intelligent equipment that provides recipe namespace capability, for example, would be an *agent* with a recipe namespace *resource*, as well as a computer platform, operating system, and electro-mechanical components, some or all of which represent other types of *resources*. An *agent* may be a component of another *agent*, and it may also contain other *agents* as components. For example, a cluster module is a component of a cluster and may itself contain intelligent subsystems as components.

Agents may, in some cases, share certain *resources* with other *agents*. An example is a docking station that connects two clusters.

Services defined by RMS may be provided at various levels within the factory. The generic term *agent* may be applied at any of these levels as appropriate. Typical *agents* that use and provide Recipe Management services include *equipment*, clusters, cluster modules, cells, and independent *recipe namespace* servers[9] . The term *agent* applies equally well to each of these.

A **service resource** is a set of services within a particular area of specialization. *Service* resources of interest to RMS are the **recipe namespace resource**, the **recipe execution resource**, and the Object Services Resource. The *service resource* object, illustrated in Figure 7.1, allows a group of message services defined by a service standard (such as RMS) to be represented by a single object, one of the subtypes of the *service resource*. However, this concept is introduced for clarification only. The *service resource* object is not a *standardized object*.

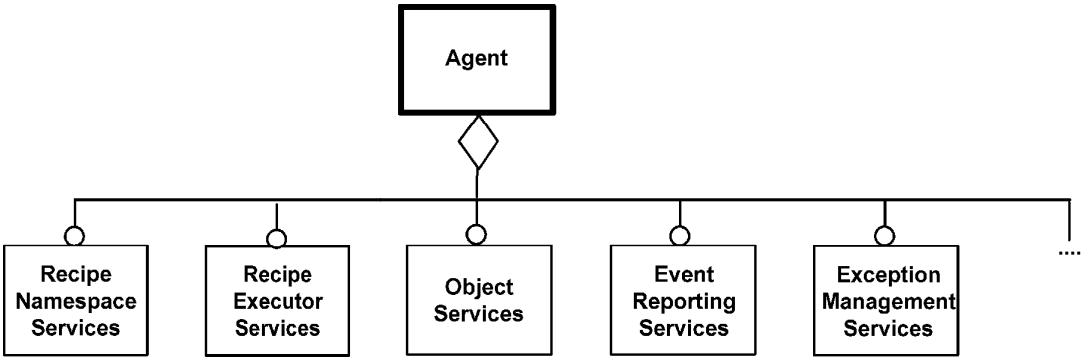


Figure 7.1
Examples of Service Resource Subtypes

An **agent** is introduced as a *standardized object* that provides one or more *service resources*. Figure 7.1 shows an example of an *agent* composed of different *service resources*. Also, an *agent* may *supervise subordinate agents* or be *supervised by a superior agent*, each of which in turn will possess their own *service resources*.

9 A recipe namespace server is an agent whose primary function is to provide recipe namespace capabilities.

Agents interact with one another collaboratively and/or hierarchically, through the *service resources* that they provide, to perform work in the factory.

7.2 *RMS Resources* — The service resources defined by RMS are the **recipe namespace resource** and the **recipe execution resource**.

A *recipe namespace resource* consists of the set of message services defined in Section 12, corresponding to the *namespace* operations defined in Sections 8 and 9. All services required for operations designated as required are *fundamental* and shall be provided for an RMS-compliant *recipe namespace resource*.

A **recipe execution resource** consists of the set of messages defined in Section 14 corresponding to the *recipe executor* operations defined in Section 11. All services required for operations designated as required are *fundamental* and shall be provided for an RMS-compliant *recipe execution resource*.

7.3 *Agent Attributes* — Table 7.1 defines the *public attributes* of *agents* that are required for recipe management.

Table 7.1 Agent Attribute Definition

Attribute Name	*Definition*	*Access*	*Reqt*	*Form*
ObjType	Agent object type.	RO	Y	Text: "Agent"
ObjID	The agent's name, assigned by an *authorized user.*	RO	Y	Text.

8 Recipe Management Operations

There are two important kinds of operations involving recipes within a *namespace*. Those that change a recipe's attributes or body are called **recipe management** operations. Those that affect the set of recipe *identifiers* within a *namespace* are part of **recipe namespace management** operations. A few operations qualify as both *recipe management* and *namespace management* and are discussed in their different aspects under both topics.

A third type of operation is informational only and requires reading but not changing recipe attributes. Operations of this type that require knowledge of the attributes of the *namespace* or of more than one recipe are discussed in Section 9.

This section describes **recipe management** operations.

Requests for *recipe operations* are always directed to the *namespace* where the recipe is stored. *Recipe management* operations are delegated by the *namespace manager* to the *namespace segment*, which is considered as the **recipe manager**. This is invisible for a *centralized namespace* and explicit for a *distributed recipe namespace*.

Operations may be invoked by the *operator* or through *namespace* services defined in Section 12. In many cases, service scenarios consist of a single message request from the service user and a corresponding response from the *namespace manager*. This case is illustrated in Section 12.2, Figure 12.1. Operations requiring additional messages are discussed in Section 8.2.

The service user is responsible for proper authorization of the user prior to requesting an operation that is restricted to authorized users through RMS services.

8.1 *Recipe Lifecycle* — A typical production recipe goes through various stages in its development within a *namespace*. These stages are shown below in a typical order through their associated operations. The attributes concerning the body are provided by the initiator of the operation.

create — The **create** operation enters a recipe *body* into the *namespace*.

Editing is expected, but not required, to be performed outside the *namespace.* Where provided by the *manager*, the same requirements concerning *creating* or *updating* a recipe are to be followed.

update — A recipe is **updated** when a body (typically, a modification of the original body) is entered into the *namespace* to replace the *body* of an existing recipe that is not **write-protected**. The attributes concerning the body are provided by the initiator of the operation.

verify — The **verify** operation is used to build a recipe. Checks for semantic correctness may also be performed at this time but are not required. The recipe *body* is read and checked for syntactical correctness, and all *external references* and *variable parameter definitions* are collected. The *verification* operation may be delegated to a *recipe executor*, which returns the required information.

write-protect — At some point during its development, an *authorized user* needs to be able to prevent the accidental deletion or modification of a recipe and requests to have it be **write-protected**. A **write-protected (read-only)** recipe may not be *updated*, **renamed, deleted, unlinked,** or **relinked**.

link — The **link** operation is used to signify that the recipe is ready for *execution*. If the recipe has *external references*, *linking* also builds a *linked recipe set* by collecting the *external references* and *variable parameter definitions* and saving them for quick access

in the *main* recipe's attributes. *Linking* resolves all *identifiers* explicitly for recipes that are within the *namespace*.

unlink — At times it may be desirable to undo the *link* operation. The **unlink** operation *clears* those attributes set by the last *link* operation. A *write-protected* recipe may not be *unlinked*.

approve — The factory uses the recipe's *approval level* to indicate the level of its authorization. For example, *approval levels* of 1, 2, and 3 may indicate "write-protected," "authorized for engineering," and "authorized for production," respectively.

modify variable parameters — The *variable parameter definitions* of a *linked* recipe may be adjusted for a specific *executing agent* to achieve the desired result. For example, a *generic* parameter for time may be incremented repeatedly for a specific furnace until it goes out of range, which indicates that it needs to be cleaned. If the recipe did not previously have an *agent-specific dataset* for this *agent*, this operation causes one to be created.

certify — An *authorized user* (typically by the process engineer who developed the recipe) assigns it a **certification-level** to indicate that it achieves the desired result on a specific *executing agent*. The factory may require certain **certification levels** for production.

de-certify — the *certification level* of the recipe is reset to zero.

unprotect — Before a recipe can be *updated, deleted, renamed, re-linked,* or *unlinked*, the recipe must be *unprotected*.

delete — **Deleting** a recipe causes it to be removed from the *namespace*.

8.2 *Description of Operations* — Operations that require additional messages are of two types: operations that may be performed on more than one recipe (such as **certify recipe**) and operations that may require interactions with a *recipe executor* to complete (**verify recipe**). The scenario for *verify recipe* is shown in Section 9.4.7.

The remaining operations that require additional messages are invoked with the *namespace* service RMNAction, where multiple recipes are specified for the operation. Operations such as these may require more time to complete. The initial response to the message service request only indicates the intent to perform the operation. In this case, the scenario is illustrated by Figure 8.1 and Figure 12.2, Section 12.2. The *namespace manager* performs the operation for each recipe, in the order specified, and upon the completion of each operation, sends the notification message RMNComplete with the results for that operation. (See Sections 12.15 and 12.17 for additional details.)

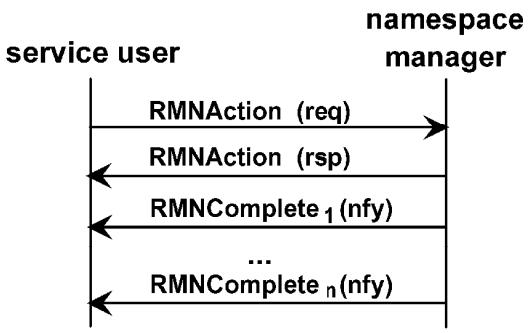


Figure 8.1
Message Flow with Completion Notification

Recipe management operations are categorized into three types: **recipe origination, recipe building,** and **recipe authorization**.

Recipe origination includes those operations that create or modify an entire recipe. **Recipe building** includes the *verification, link, unlink,* and *modify variable parameter* operations. **Recipe authorization** includes the *write-protect,* **unprotect, approve,** and **certify** operations.

Except for those operations that only provide information, *recipe management* operations change the state of the recipe. Section 8.3 contains the complete state model, and a table of transitions is given in Table 3.1. Substates of this model are provided for illustration in sections defining the operations that affect these substates.

8.2.1 *General Requirements* — The *generic length* and *timestamp* attributes AttrLength and AttrChgTime shall be updated whenever any other *generic* attribute changes value.

An *agent-specific dataset* for a specific *agent* exists only when a non-required attribute is given a *non-default value*, such as whenever a recipe is *certified* for a given *agent*. If the non-required attributes are all *cleared*, the *agent-specific dataset* is **removed** and no longer exists. Otherwise, the *agent-specific length* and *timestamp* attributes for an *agent-specific dataset* shall be updated whenever one of its attributes changes value.

8.2.2 *Recipe Origination* — A recipe may be originated by the *create recipe* operation and by the *copy recipe* operation. The *copy recipe* operation creates a duplicate of an original recipe and assigns it a new *identifier*. All

of the attributes of the original are copied directly. These operations are included in *Namespace Management* in Section 9.4.1.

8.2.2.1 *Create Recipe* — A recipe is **created** when a recipe *body* is first entered into a *namespace* and assigned a new *identifier*. The recipe's *mandatory attributes* are set. All other *attributes* take on their default values. The values for the *attributes* BodyLength, *EditTime*, and EditedBy are required to be provided by the initiator of the *create* operation. The attribute BodyFormat is also provided at this time if it is not in source form (i.e., if BodyFormat has a *non-default* value).

A newly *created* recipe has an active state model (Figure 8.7). The recipe is in the UNVERIFIED, UNLINKED, UNAPPROVED, UNPROTECTED, and UNCERTIFIED states.

The *create recipe* operation is invoked with the RMNCreate service.

8.2.2.2 *Update Recipe* — The **update** operation is identical to that of *create* except that an *unprotected* recipe with the specified *identifier* already exists, and the new *body* replaces the existing *body*. Attributes concerning the *body* are provided by the initiator of the request, as for the *create* operation. All *non-mandatory attributes* are *cleared* (reset to their *default values*). Any existing *agent-specific datasets* are discarded.

The *update recipe* operation is invoked with the RMNUpdate service.

8.2.3 *Recipe Building* — Building a recipe is a two-step process. First the recipe is *verified* and then *linked*.

8.2.3.1 *Verify Recipe* — The **verify operation** is the only time it is necessary to parse the contents of the recipe *body* until the recipe is *executed*. The primary purpose of this process is to ensure that the syntactical and lexical structure of the *body* is correct. One or more checks for semantic correctness may also be performed as part of the *verification* operation, but this is not required.

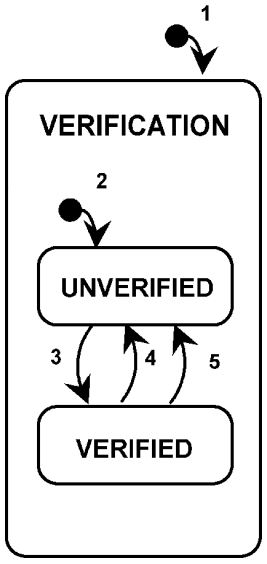
Actual *verification* is performed by a member agent's **recipe executor** at the request of a *namespace manager*. For this reason, this operation is also discussed under *namespace management* in Section 9.4.7.

Recipes may be stored in a *namespace* in an incomplete or unfinished form. For recipes in *source form*, the *verification* procedure shall be performed only at the request of the *user*. It shall not be performed automatically.

There are four attributes affected by the *verification* operation: Verified, EstRunTime, ExtRef, and Parameters.

The boolean *generic attribute* Verified is used to indicate the recipe's state with respect to this procedure. Verified is *cleared* when a recipe is first created and whenever it is updated. Verified is set TRUE only when the recipe passes the *verification* by a *recipe executor*. A recipe is considered to be **verified** if, and only if, the Verified *attribute* is TRUE. Figure 8.2 illustrates the recipe's VERIFICATION state. For a description of the transitions, see Table 8.1 in Section 8.3.

Figure 8.2



Verification State Model

The values for the attributes EstRunTime, ExtRef, Parameters, and VerificationID are returned (where set) by the *recipe executor* when the *verification* has been successful. If the operation is unsuccessful, they shall be *cleared*.

ExtRef contains a list of all *external references* found within the *body* of the recipe by the *recipe executor*. These references may be explicit or implicit, leaving the *class* and/or *version* unspecified.

The *verify recipe* operation is invoked with the RMNAction service.

8.2.3.2 *Link Recipe* — The **link operation** is a required *operation* that may be requested at any time by the *operator* or *supervisor*. Successful completion of the *link operation* indicates that the recipe is ready for *execution*. For single-part recipes without *variable parameters*, *linking* consists only of *setting* the *generic*

attribute Linked to TRUE. Figure 8.3 illustrates the LINKAGE state of the recipe. For a description of the transitions, see Table 8.1 in Section 8.3.

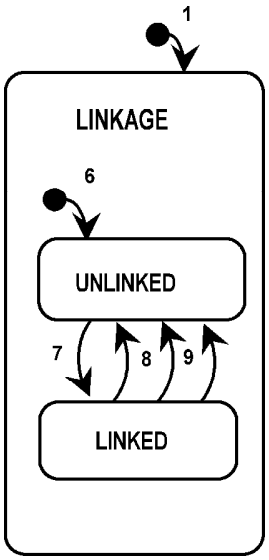


Figure 8.3
Link State Model

For multi-part recipes, this *operation* also collects *external references* and *variable parameter definitions* into the LinkList and LinkParam attributes, to be used by the *recipe executor* at run-time.

External references contained within multi-part recipes are not required to specify the *class* of the *subrecipe* when the *class* is the same as that of the *parent* recipe. Where *class* is omitted in a recipe *identifier* in the ExtRef attribute, the *link* operation is responsible for adding the *class* of the *parent recipe* in the *identifier* added to the LinkList parameter, as the *parent/child* relationship cannot be derived from the final contents of LinkList.

External references are assumed to refer to recipes within the same *namespace* and are not required to give explicit *versions* unless they specify a different *namespace*. This ensures that the *user* will be able to *link* together the "best choice" *subrecipes*. The exact value of *versions* of such references is determined only at the time the *main* recipe is *linked*.

The **link** operation is performed on a *main* recipe. That is, it only modifies *attributes* of the *main* recipe. It resolves all *external references* within that recipe and within any of its *subrecipes* according to well-defined rules for determining default *versions* at the time of the *link*. At the same time, *variable parameter definitions* are collected. The resulting explicit references are placed in the LinkList *attribute*, *variable parameter definitions* are placed in LinkParam, the ApprovalLevel attribute is *cleared*, the Linked attribute is set to TRUE, and the *main* recipe is then said to be **linked**.

Subrecipes are not affected by this operation. *Subrecipes* of a *main* recipe have no knowledge of parents or of linkages. As a result, it is possible to *delete* or change a *subrecipe* with an unintended detrimental impact on a *linked recipe set*. Factory policy may designate certain levels of *approval* to mean "this recipe is used by (*linked* into) one or more *protected* recipes."

For multi-part recipes, *linking* starts at the *main recipe* and works through all chains of referenced *subrecipes* to determine the complete set of *identifiers* that will comprise the recipe as a whole. The *link* operation is a mechanical procedure that may be performed at any time.

Successful *linking* implies that all referenced *subrecipes* have been located and parsed for further references until all references are exhausted. The *link* operation shall fail when any recipe or *subrecipe* within the namespace either is *unverified* or cannot be located.

If the same *parameter name* is used to *define* a *parameter* in more than one recipe of a set of recipes that are *linked* together, the *name* shall represent the same *parameter* and have the same *parameter definition* in all recipes in the set, to avoid ambiguity. *Parameters* with the same *parameter name* and differing *definitions* shall cause, at a minimum, a warning to the user when the *link operation* completes.

For purposes of comparison of results, the order of references in LinkList and LinkParam at the completion of the *link* operation shall conform to the results when the following sequence is used:

1. Copy the contents of ExtRef from the *main* recipe to LinkList, resolving *class* and *version* to each recipe *identifier* as needed. For support of *variable parameters*, also copy the contents of Parameters to LinkParam. If ExtRef is *empty*, then LinkList is also *empty* and the parsing process is complete.
2. If LinkList is not *empty*, begin with the first reference in LinkList as the **link target reference**.
3. If the *link target reference* lies within the same *namespace*, resolve the *class* and *version* if necessary. (Note: the referenced recipe must be already present within the *namespace* at the time the *link* is performed, or else the *link* fails.) If the *link target reference* specifies a different *namespace*, go to Step (6) — the *link* operation of this chain terminates (without error) without an attempt to locate the actual recipe or its ExtRef *attribute*. If the

recipe is located, then it is called the **link target recipe**.

4. Determine the contents of the Verified attribute of the *link target recipe*. If Verified is FALSE, the *link* operation fails immediately.

5. Determine the contents of the ExtRef *attribute* of the *link target recipe*. Remove any references that duplicate those already contained in LinkList, resolve *class* and *version* as needed, and append the result to LinkList. If *variable parameters* are supported, also determine the contents of Parameters in the *link target recipe*, remove *parameters* already defined in LinkParam, and append the result to LinkParam.

6. Set the *link target reference* to the next reference in LinkList and repeat steps (3) through (6) until all references in LinkList have been processed.

For recipes with no *external references*, the LinkList list will be *empty*. Similarly, for recipes with no *variable parameters*, the LinkParam list will be empty.

The effect of allowing incomplete recipe *versions* to be specified within a recipe *body*, and determined only when the *main* recipe is *linked*, means that a *linked recipe set* (the set of recipes *linked* together) produced on one occasion may not be the same as those produced on a different occasion. Therefore, it is necessary to protect the contents of the LinkList *attribute* from inadvertent change.

For this reason, the *link* operation cannot be performed on a *read-only* recipe with the Linked *attribute* already set from a previous *link*. Any attempt to *link* an already *linked read-only* recipe either shall be denied or a new copy of that recipe shall be generated with a new *identifier*, which may then be automatically *approved* and *linked*.

The *link recipe* operation is invoked with the RMNAction service.

8.2.3.3 *Unlink Recipe* — A *linked* recipe that is not *write-protected* may also be *unlinked*. The *unlink* operation *clears* the *generic attributes* Linked, LinkList, and LinkParam. If the recipe has *agent-specific datasets*, the attributes Certified and AgentSpec LinkParam are *cleared*. If no other non-required *agent-specific* attributes have a *non-default value*, the *agent-specific dataset* is removed.

The *unlink recipe* operation is invoked with the RMNAction service.

8.2.3.4 *Modify Variable Parameters* — The recipe *attributes* Parameters, LinkParam, and AgentSpec_LinkParam each consist of a list of **parameter definitions**. Each *parameter definition* contains the *parameter name, parameter initial value,* and the *parameter restriction* (if any) specified for that *parameter* by a formal definition within the recipe body.

In order to "tune" a recipe so that it produces the same result on all *executing agents* of the same type, it may be necessary to provide a different *initial value* or a different *parameter restriction* for one or more *parameters* for individual *executing agents*. The *agent-specific* attribute AgentSpec_LinkParam is used to provide this capability.

AgentSpec_LinkParam is an optional *agent-specific attribute* that contains a list of alternate *parameter definitions* for one or more of the *variable parameters* included in the *definitions* in the *generic attribute* LinkParam of a *linked* recipe. A special editing facility may be provided to allow an *authorized user* to **add**, **delete**, or **modify** *parameter definitions* to AgentSpec_LinkParam.

Parameter definitions initially are added individually to AgentSpec_LinkParam by copying the *definition* for that *parameter* from LinkParam. The *initial value* or *restriction* then may be modified by the user, subject to the absolute restrictions, such as minima and maxima, imposed by the *executing agent's* supplier. The *initial value* may be changed to any value within the *parameter domain*.

Any modification of the *restriction* shall cause the *certification level* to be *cleared* in the Certified *attribute*. This is required because of the potential impact on fab operations of a change in the *restriction*.

The UNITS in the *restriction* of a *numeric parameter* may be modified within constraints imposed by the use of the *parameter* and the *executing agent's* supplier. For example, a two-byte unsigned integer named WaitTime that is used to set a variable time delay period may permit units of either "min" (minutes) or "s" (seconds). It is desirable, but not required, that suppliers of *executing agents* support different options for specifications of UNITS. The possible options for each potential *variable parameter* shall be documented by the supplier.

The **modify variable parameter** operation consists of **adding**, **deleting**, or **modifying** a specific *parameter definition* to a list of *definitions* in AgentSpec_LinkParam for a specific *agent*.

A *definition* is **added** when it is copied directly from the LinkParam *attribute* to the AgentSpec_LinkParam *attribute* without modification. A *definition* shall only be *added* if there is currently no *definition* for a *parameter* with the same *parameter name* in the AgentSpec_LinkParam *attribute*.

A *definition* is **deleted** when it is completely removed from the list of *definitions* in AgentSpec_LinkParam.

A *definition* is **modified** when the *value* or *restriction* of an existing *definition* is replaced in AgentSpec_LinkParam.

The *modify variable parameters* operation is invoked with the RMNVarPar service.

8.2.4 *Recipe Authorization* — Recipe authorization operations include those that allow an *authorized user* to change the value in the *generic* attribute Approval-Level or the *agent-specific* attribute Certified.

The *de-certify* operation is invoked with the service RMNAction.

8.2.4.1 *Approve Recipe* — Recipe management provides two different methods of controlling how a recipe is applied: through the *generic* attribute ApprovalLevel and through the *agent-specific* attribute Certified. The recipe's **approval level** (the value contained in ApprovalLevel) is also used to protect it from change.

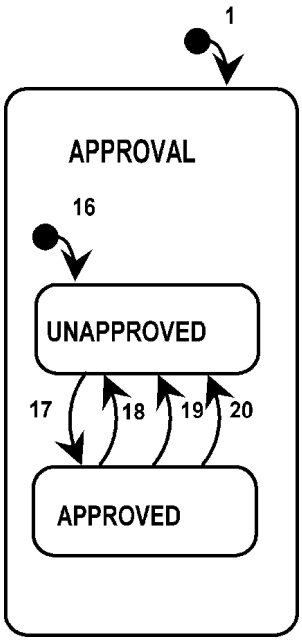
A recipe goes through different stages during its lifecycle. After a recipe has been initially created, it will typically go through a dynamic period while it is tested and adjusted until it produces the desired results.

ApprovalLevel is an unsigned integer used to designate the different stages in a recipe's life cycle. It is set to zero (its *default value*) whenever a recipe is *created*, *updated*, or *linked*.

The **approval** operation allows an *authorized user* to set ApprovalLevel to a non-zero value. Otherwise, it may not be set externally.

A recipe is said to be **approved** whenever Approval-Level is non-zero. An example of a factory's implementation of *approval levels* is given in the appendix, Section 8.3. Figure 8.4 illustrates the recipe's APPROVAL state. For a description of the transitions, see Table 8.1 in Section 8.3.

Figure 8.4
Approval State Model



A *subrecipe* may be *approved* independently from any recipes which reference it.

A *linked* recipe may not be *approved* to a level higher than the lowest *approval level* of any of its *subrecipes*. Therefore, all *subrecipes* referenced in the *attribute* LinkList of a *linked* recipe must be located prior to granting an *approval level* other than zero. The ApprovalLevel *attribute* of the *subrecipes* must be increased to a value equal to, or greater than, the required *level* by an *authorized user* before the higher *approval level* of the *linked recipe* is accepted.

The restriction on the *approval level* of the *linked* recipe requires the *user* to purposefully change the *approval level* in order to protect the entire *linked recipe set* as a unit from unexpected change, as it is by the *main recipe identifier* that the recipe as a whole will be known. It should be noted that *approval* of a *linked* recipe is not the same as approval of the individual parts, as a *subrecipe* which is appropriate in one recipe may be inappropriate in another.

The *approve recipe* operation is invoked with the RMNAction service.

8.2.4.2 *Certify* — The **certify** operation sets the value of Certified for a specific *executing agent* to a non-zero value specified by an *authorized user*. This operation affects no other attributes (except the *length* and *timestamp* attributes of the affected *agent-specific dataset*).

A recipe may be syntactically and procedurally correct but, due to the differences between different installations of *agents* with access to a *namespace*, may give different results on these different installations. Recipe **certification** signifies that a *linked* recipe produces the desired results on a specific installation.

A recipe is considered **certified** for *agent* if the **certification-level** contained in the *agent-specific attribute* Certified is non-zero. Otherwise, the recipe is *uncertified*. A *certified* recipe is *de-certified* when the *certification-level* is set to zero.

A recipe may be *certified* and *de-certified* only by an *authorized user*. Figure 8.6 illustrates the recipe's CERTIFICATION states. For a description of the transitions, see Table 8.1 in Section 8.3.

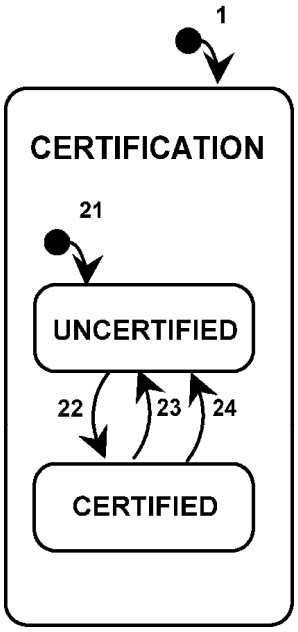


Figure 8.6
Certification State Model

Recipes with *variable parameters* may be *certified* for specific *values* and *restrictions* of some of these variables. If so, such values are stored in the AgentSpec_LinkParam *attribute*.

Only recipes with the Linked *attribute* set to TRUE may be *certified*.

The *certify recipe* operation is invoked with the RMNAction service.

8.2.4.3 *De-certify* — The **de-certify** operation clears the Certified attribute for a specific *agent* at the request of an *authorized user*.

A recipe may need to be *de-certified* after major maintenance has been performed and later *re-certified* only after testing its results.

8.2.5 *Recipe Protection* — The *namespace* attribute RecipeReadOnlyLevel is used as a threshold to govern the level of approval required for individual recipes to be protected. A recipe is **protected** when the value in its ApprovalLevel attribute is equal to, or greater than, RecipeReadOnlyLevel. All recipes are protected whenever the namespace's RecipeReadOnlyLevel attribute is equal to zero. Figure 8.5 illustrates the recipe's PROTECTION states. For a description of the transitions, see Table 8.1 in Section 8.3.

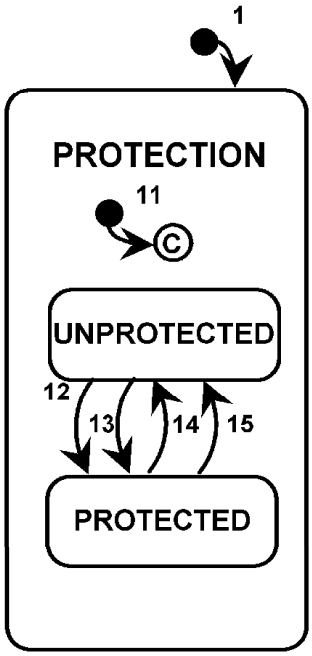


Figure 8.5
Protection State Model

If the value of RecipeReadOnlyLevel is n, then recipes with *approval levels* greater than, or equal to, n are ***read-only***. The body of a *read-only* recipe may not be changed in any way, including by over-writing or deleting. The *identifier* of a *read-only* recipe may not be renamed. A *linked* recipe that is *read-only* may not be *re-linked*.

The *recipe protect* operation sets the value of ApprovalLevel to the value in RecipeRead OnlyLevel. For *linked* recipes with *subrecipes*, the operation is denied if the resulting *approval level* for the main recipe would be higher than any *subrecipe*, as described in Section 8.2.4.

If the value in RecipeReadOnlyLevel is zero, all recipes within the *namespace* are automatically *write-protected*, regardless of the support for the ApprovalLevel *attribute*.

The *protect recipe* operation is invoked with the RMNAction service.

8.2.6 *Unprotect* — The *read-only* status of a recipe may be changed either by changing its *approval level* to a value less than the value of RecipeReadOnlyLevel or by increasing the value in RecipeReadOnlyLevel.

NOTE: If RecipeReadOnlyLevel is zero, all recipes are write-protected regardless of the value in approval-level.

A protected recipe may be changed to **unprotected** at the request of an *authorized user*. This operation *clears* the ApprovalLevel attribute.

The unprotect recipe operation is invoked with the RMNAction service.

8.2.7 *Informational Operations* — Object Services are used to request the current value of one or more recipe attributes and to set one or more values. Certain *attributes* have **restricted access** and may not be set through RMS *services*. These *attributes* are identified as "RO" (read-only) in Tables 3.1 and 3.2 in the column labeled "Access" in Section 3.4.2. When attributes identified as "RW" (read-write) are set through Object Services, the appropriate attribute *length* and *timestamp* attributes shall be updated appropriately.

In addition to Object Services, the **get descriptors** operation provides important information.

8.2.7.1 *Get Recipe Descriptors* — The **get recipe descriptor** operation returns the *descriptor* of a specified recipe: *the generic descriptor*, the *body descriptor*, and the *agent-specific descriptors* of any existing *agent-specific datasets*.

A *recipe descriptor* may be used to determine if two or more recipes are identical or which is most recent.

The *get recipe descriptors* operation is invoked with the RMNGetDescriptor service.

8.3 *Recipe State Model* — An existing recipe has different states of interest to RMS. These are shown in Figure 8.7. The Recipe Available State Model in Figure 8.7 combines the separate models for VERIFICATION, APPROVAL, PROTECTION, LINKAGE, and CERTIFICATION as AND substates of RECIPE AVAILABLE.

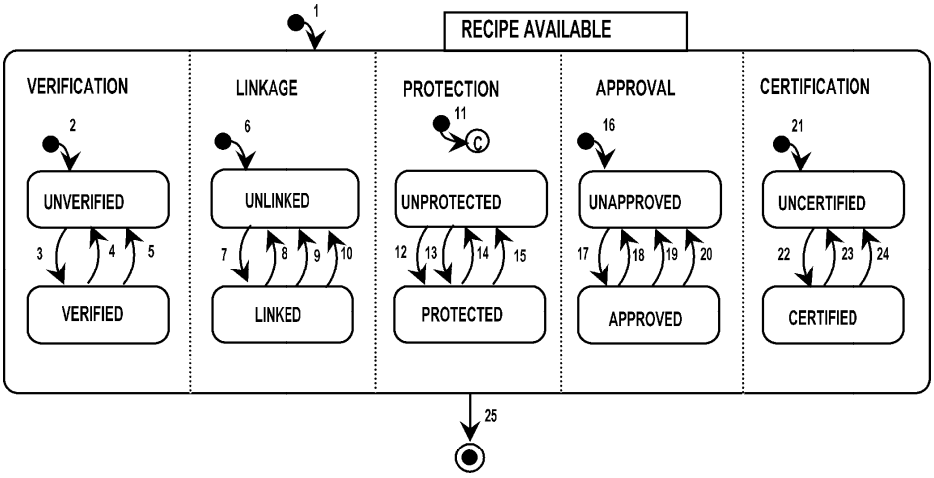


Figure 8.7
Recipe Available State Model

These states have been discussed in previous sections and are associated with one or more of the recipe's *attributes*, as follows:

VERICATION: Verified,

APPROVAL: ApprovalLevel,

PROTECTION: the interaction of ApprovalLevel and the *namespace attribute* RecipeReadOnlyLevel,

LINKAGE: Linked, and

CERTIFICATION: Certified.

The table of transitions is given in Table 8.1.

Table 8.1 Table of Transitions

#	*Current State*	*Trigger*	*New State*	*Action*	*Comment*
1	(entry to Recipe Available)	Recipe is created.	Recipe Available.	Initialize generic attributes.	None.
2	(default entry to VERIFICATION)	Recipe is created.	UNVERIFIED	Verified reset.	Newly created recipe is unverified.
3	UNVERIFIED	Requested verification operation is successful.	VERIFIED	Set Verified to TRUE.	Verification is performed by the recipe executor.
4	VERIFIED	Requested verification operation, on previously verified recipe, fails.	UNVERIFIED	Reset Verified.	Verification is performed by the recipe executor.
5	VERIFIED	Recipe is updated.	UNVERIFIED	Reset Verified.	Updated recipe must be *re-verified*.
6	(default entry to LINKAGE)	Recipe is created.	UNLINKED	Reset Linked, LinkList, and LinkParam.	Newly created recipe is *unlinked*.
7	UNLINKED	*Authorized user* requested *link* operation is successful.	LINKED	Set Linked to TRUE. Set LinkList and LinkParam.	Single-part recipes set Linked only.
8	LINKED	Recipe is *updated*.	UNLINKED	Reset Linked, LinkList, and LinkParam.	Single-part recipes set Linked only.
9	LINKED	Recipe is *re-verified*.	UNLINKED	Reset Linked, LinkList, and LinkParam.	Single-part recipes set Linked only.
10	LINKED	*Authorized user requests the* unlink operation.	UNLINKED	Reset Linked, LinkList, and LinkParam.	Single-part recipes set Linked only.
11	(default entry to PROTECTION)	Recipe is created.	If RecipeRead-OnlyLevel is zero, new state is PROTECTED. Otherwise, it is UNPROTECTED.	None.	All recipes are *read only* when RecipeReadOnlyLevel is zero.
12	UNPROTECTED	*Authorized user* request to *protect* the recipe.	PROTECTED	Set ApprovalLevel to the value in RecipeReadOnly-Level.	Recipe is *protected* when *approval level* = RecipeReadOnlyLevel.
13	UNPROTECTED	Authorized user sets RecipeReadOnlyLevel to value greater than or equal to *approval level*.	PROTECTED	None.	Recipe is *protected* when *approval level* greater than or equal to RecipeReadOnlyLevel.
14	PROTECTED	*Authorized user* sets *approval level* to a value less than RecipeReadOnlyLevel.	UNPROTECTED	None.	Recipe is *unprotected* when *approval level* less than RecipeReadOnlyLevel.
15	PROTECTED	*Authorized user* sets RecipeReadOnlyLevel to a	UNPROTECTED	None.	Recipe is *unprotected* when *approval level* less than

		value greater than *approval level*.			RecipeReadOnlyLevel.
16	(default entry to APPROVAL)	Recipe is created.	UNAPPROVED	Reset ApprovalLevel.	Newly created recipe is *unapproved*.
17	UNAPPROVED	*Authorized user* sets non-zero *approval level*.	APPROVED	Set ApprovalLevel as specified by the user.	User assigns *approval level* to a specific value.
18	APPROVED	*Authorized user* sets *approval level* to *zero*.	UNAPPROVED	Reset ApprovalLevel.	User assigns a zero value as *approval level*.
19	APPROVED	Recipe is *linked*.	UNAPPROVED	Reset ApprovalLevel.	User is required to specifically approve a *newly linked recipe set*.
20	APPROVED	Recipe is *updated*.	UNAPPROVED	Reset ApprovalLevel.	A modified recipe is *unapproved*.
21	(default entry to CERTIFICATION)	Recipe is *created*.	UNCERTIFIED	None.	Newly created recipe has no *agent-specific attributes*.
22	UNCERTIFIED	*Authorized user* assigned non-zero *certification level*.	CERTIFIED	Create *agent-specific dataset* if necessary. Set Certified to specified value.	User assigns *certification level*.
23	CERTIFIED	*Authorized user* assigned zero *certification level*.	UNCERTIFIED	Reset Certified.	*Agent-specific dataset* not required if there are no *agent-specific attributes*.
24	CERTIFIED	*Authorized user* changes *agent-specific variable parameter restrictions*.	UNCERTIFIED	Reset Certified.	A change in restrictions affects performance of recipe.
25	CERTIFIED	*Recipe* is *unlinked*.	UNCERTIFIED	Reset Certified attribute.	Agent-specific dataset not required if there are no agent-specific attributes.
26	RECIPE AVAILABLE	Authorized user deletes unprotected recipe.	(Undefined recipe no longer exists.)	Storage occupied by recipe becomes available.	None.

8.4 *Table of Operations* — Table 8.2 lists the *recipe management* operations in the order presented in this section. The column labelled "Rqmt" is used to indicate those operations that are required for *fundamental compliance* to RMS.

Table 8.2 Recipe Management Operations

Operation	*Description*	*Rqmt*
create recipe*	A new recipe body is entered into the namespace.	Y
update recipe*	The body of an existing recipe is replaced.	Y
verify recipe	Check the syntax or format of a recipe body for correctness.	Y
link recipe	A *main* recipe is *linked*.	Y
unlink recipe	An *unprotected linked* recipe is *unlinked*.	N
modify variable parameters	A *linked* recipe's parameter definitions are modified within an *agent-specific dataset*.	N
approve recipe	Set the recipe's *approval level* to a non-zero value.	N
protect recipe	Set the recipe's *approval level* to the value in RecipeReadOnlyLevel.	N
unprotect recipe	Set the recipe's *approval level* to zero.	N
certify recipe	A linked recipe's certification level is set to a non-zero value.	N
decertify recipe	A linked recipe's certification level is set to zero.	N
get recipe descriptor	A recipe's descriptor is requested.	Y

*This operation is also covered under Namespace Management Operations.

9 Namespace Management Operations

Namespace management operations include operations that affect the *namespace* itself in some way and those operations that provide information about the *namespace* or its *manager*, or that require knowledge about more than one recipe.

Namespace operations defined in RMS are presented in groups of similar functionality:

- *operations on the* namespace *(create, delete, and rename* namespace*),*
- *operations that provide information about the* namespace *or its recipes (get available storage, check recipe status, and get best version), and*
- *operations on recipes that affect the set of recipe identifiers within the* namespace *and/or moving a recipe as a whole (create, delete, store, retrieve, copy, and rename recipe),*
- *operations that always require interactions with a* recipe executor *(*verify*, download, and upload recipe).*

Namespace management operations may be invoked through *namespace* services defined in Section 12. In most cases, service scenarios consist of a single message request from the service user and a corresponding response from the *namespace manager*. This case is illustrated in Section 12.1.

Scenarios are shown only for operations that require additional messages. These operations are one of two types: operations that may be performed on more than one recipe (such as **delete recipe**) and operations that require interactions with a *recipe executor* to complete.

Operations such as these may require more time to complete. The initial response to the message service request only indicates the intent to perform the operation. The *namespace manager* informs the service user of the completion of each individual operation by sending the notification message RMNComplete. (See Section 12.1 for more detail.)

9.1 *Applications of Object Services* — A *manager* shall comply with SEMI E39 (Object Services Standard (OSS): Concepts, Behavior, and Services) specifications for *fundamental requirements* and with the requirements for Filters and Owner Objects.

9.1.1 *Object Specifiers* — The "owns/owned by" relationship is used by OSS to define the object specifier used for scope. Recipes within a *namespace* are *owned* by the *namespace* in which they reside. The *namespace* in turn is owned by its *manager*. The *agent* that provides the storage and services for a *centralized namespace* owns both the *manager* and the *namespace*.

An object specifier has the form of:

"$type_1 : id_1$>…$type_n : id_n$>"

where "$type_i$" and "id_i" represent the object type and object identifier, respectively, of the ith object instance in the sequence, and where each object is owned by the preceding object in the sequence and is the owner of the succeeding object.

A **namespace specifier** is an object specifier applied to *namespaces*. A **recipe specifier** is an object specifier applied to recipes.

Object types in the object specifier may be omitted where they may be otherwise determined. For the *recipe specifier*, when omitted, they are determined by their relative positions, with the recipe identifier in the final position, preceded by a *namespace identifier*. Additional *identifiers* preceding that of the *namespace* are those of *agents*.

An example of a *namespace specifier* for a namespace "NS-MOM" owned by an *agent* "Etch01" would be "Agent:Etch01>RNS:NS-MOM>" or (where object types can be otherwise determined) "Etch01>NS-MOM". A recipe specifier for recipe "/PROCESS/ABC;5" stored in NS-MOM would be "Agent: Etch01>RNS:NS-MOM>/PROCESS/ABC;
5>".

Where the *manager* is to be used instead of the *namespace*, the object type of the *manager* must be included.

A recipe in a *namespace* also owns its components. An *agent-specific attribute* is accessed through the recipe owner.

9.1.2 *Required Object Services* — A *manager* shall support operations for Get Attributes and Set Attributes for the *attributes* of the *namespace*, the recipes within the *namespace*, and the *manager* itself. For a *shared namespace*, access to different *agent-specific datasets* shall be supported.

When a recipe's *attributes* are changed through the Set Attributes operation, the appropriate attribute *timestamp* and *length attributes* shall be updated as well.

A *manager* shall support the Get Type and Get Attribute Name operations for the object types of *namespace*, recipe, recipe *components*, and *manager*.

9.2 *Namespace Operations* — This section describes operations that are performed on a *namespace*.

9.2.1 *Create Namespace* — The **create namespace** operation is used to define a *namespace* and assign a *name* to be used as its *identifier* ObjID. The *name* is assigned only by an *authorized* user, except that a *name*

of "Default" is prohibited. Once *created*, the *namespace* shall be ready to accept recipes.

This is an optional capability that is not required if the *owner agent* that provides the *namespace* capabilities also provides an installed *default namespace* (see Section 6) that cannot be deleted. In this case, a means of recreating the *namespace* shall be provided in the event the *namespace* becomes damaged.

The *create namespace* operation is invoked with the message service RMNCreateNS.

9.2.2 *Delete Namespace* — The **delete namespace** operation is the inverse of the *create namespace* operation. A *namespace* that is not *empty* may not be *deleted*. It is recommended that the *default namespace* of a *recipe executor* should not be *deleted*. This is an optional capability required only if the *create namespace* operation is supported.

The *delete namespace* operation is invoked by the message service RMNDeleteNS.

9.2.3 *Rename Namespace* — The **rename namespace** operation allows an *authorized user* to change the *identifier* of the *namespace*. It is recommended that the *default namespace* of a *recipe executor* should not be renamed. This is an optional capability required only if the *create namespace* operation is supported.

The *rename namespace* is invoked by the message service RMNRenameNS.

9.3 *Namespace Informational Operations*

9.3.1 *Get Available Storage* — The **get available storage** operation is used to determine the size of the remaining recipe storage capacity, in bytes. The value returned shall exclude any overhead requirements for storage of one *generic* recipe. That is, it shall be assumed that sufficient storage exists for a single recipe with a combined *generic attribute length* and *body length* less than or equal to the returned value, and ignoring possible space requirements for additional *agent-specific datasets*. This is a required operation.

The *get available storage* operation is invoked by the message service RMNSpaceInquire.

9.3.2 *Check Recipe Status* — The **check recipe status** operation checks a recipe *identifier* and returns the status for existence and *read-only* (PROTECTED or UNPROTECTED state). It also returns the next available *numeric version*. This is a required operation.

This operation may be used to determine if a given recipe *identifier* will be accepted prior to sending it to the *namespace* and to obtain an available version if the original *identifier* is used for a *read-only* recipe.

The *check recipe status* operation is invoked by the message service RMNRecStatInquire.

9.3.3 *Get Best Version* — The **get best version** operation checks for the best *default version* of a recipe with a specified *class* and *name* and for an optional specific *member agent*. If a *member agent* is specified, then in addition to the rules for selection of a default version defined in Section 3.2.3.4.1, the *version* with the highest *certification level* for that *agent* is selected. This is a required operation.

The *get best version* operation is invoked by the message service RMNVersionInquire.

9.4 *Namespace Recipe Operations* — This section describes the recipe operations that affect the set of recipe *identifiers* within the *namespace* and/or involve moving an entire recipe.

9.4.1 *Create Recipe* — A recipe is created through the **create recipe** operation when a *namespace* is sent a recipe with an *identifier*, *body*, *body descriptor*, and the *attributes* BodyFormat and EditedBy only. This operation sets the attributes AttrLength and AttrChgTime and *clears* the remaining *generic attributes*.

This capability allows a recipe body that has been created off-line to be stored in a *namespace*. It is required of any *namespace* intended for use other than as a *default namespace* for hardware-specific recipes only[10].

The *create recipe* operation is invoked by the message service RMNCreate.

9.4.2 *Delete Recipe* — The **delete recipe** operation has the effect of deleting a recipe from the *namespace*. Complete physical erasure of the recipe is not required, but the recipe is no longer considered to be stored in the *namespace*, is no longer accessible, and the storage space that it used is freed.

A *read-only* recipe may not be *deleted*.

The *delete recipe* operation is invoked by the message service RMNAction. More than one recipe may be specified by the service user. Figure 9.1 illustrates the flow of messages in this case. The *namespace manager* responds to the initial request with an intent to comply before performing any deletions. As each deletion is completed, the *manager* notifies the service user of the results using RMNComplete.

---

10 In the case of the default namespace dedicated to hardware-specific recipes, the recipes may always be created initially by the recipe executor and uploaded to namespace.

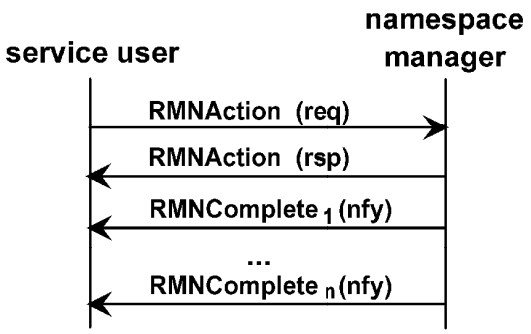


Figure 9.1

Delete Recipe Scenario

9.4.3 *Store Recipe* — The **store recipe** operation is used to store a complete recipe, including its *body, generic attributes*, and one or *more agent-specific datasets*, in a *namespace.* Note: methods of storing recipes are not specified by RMS.

Storage shall be denied if the specified recipe *identifier* is already used by an existing *read-only* recipe or if there is insufficient storage available for the recipe. Otherwise, the recipe shall be accepted into the *namespace*.

The *store recipe* operation is invoked by the message service RMNStore.

9.4.4 *Retrieve Recipe* — The **retrieve recipe** request specifies the *identifier* of a recipe. If the recipe exists within the *namespace*, the *namespace manager* returns the requested recipe. Otherwise, it shall deny the request. This is a required operation.

It is also possible to *retrieve* a recipe's *generic attributes* set to a *non-default value* and/or one or more of its *agent-specific datasets* without *retrieving* its *body*.

The retrieve recipe operation is invoked by the message service RMNRetrieve.

9.4.5 *Copy Recipe* — The copy recipe operation causes a new copy of a recipe, with a different *identifier* from the original recipe, to be created within the *namespace*. If the *identifier* for the new *copy* is already in use by a pre-existing *read-only* recipe, the namespace shall deny the request.

The *copy recipe* operation is invoked by the message service RMNCopy.

9.4.6 *Rename Recipe* — The rename recipe operation causes a recipe to be assigned a new *identifier* within the *namespace*. If the new *identifier* is already in use by a pre-existing *read-only* recipe, the namespace shall deny the request. In this case, it may suggest a new *version* according to the rules in Section 3.2.3.3.

The *rename recipe* operation is invoked by the message service RMNRename.

9.4.7 *Verify Recipe* — A *manager* is not required to understand the syntax or semantics of the recipe language of a *source* recipe or to understand the internal format of an *object form* recipe. To *verify* a recipe, the *manager* may require the services of a *recipe executor*, described in Section 6. In this case, the *manager* shall request *verification* from the *recipe executor* of one of the *agents* listed in its Members attribute and shall return the resulting status and error information to the initial requestor. (See Section 11 for more detail.) This is a required operation.

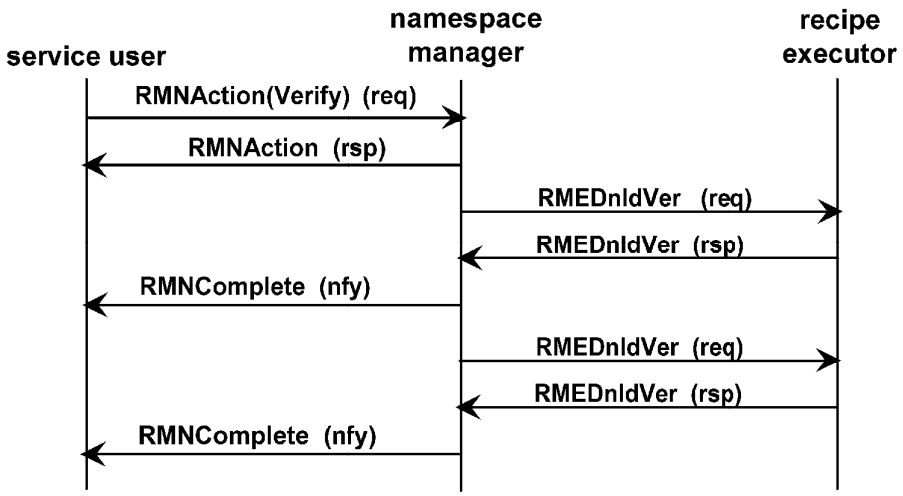


Figure 9.2
Verify Scenario

The *verify recipe* operation is invoked by the message service RMNAction. Figure 9.2 shows a typical sequence of the message flow when the *verify* operation is requested for multiple recipes. The *namespace manager* responds to the service request RMNAction (Verify) with an intent to perform the requested operations. Each recipe specified in the request is downloaded to a *recipe executor* of an *agent* listed in the *namespace* attribute Members. (Note: where the *namespace manager* and *recipe executor* are provided by the same *agent*, formal RMS services are not required for communications between the two.) In this example, the *recipe executor* responds to the *namespace manager* with an intent to comply, *verifies* the recipe, and returns the information required for the completion of the *verify* operation as described in Section 8.2.3.1. The *namespace manager*, in turn, returns the final status of the operation for that recipe in the notification message RMNComplete.

The *recipe executor* provides two operations for performing verifications, "download and verify" and "verify". The former operation does not typically *re-verify* already *verified* recipes. See Sections 11.2.1 and 11.2.2 for detail.

The *verify recipe* operation is invoked by the message service RMNAction.

9.4.8 *Download Recipe* — The **download recipe** operation causes the *namespace manager* to download a recipe to the *recipe executor* of a specified *agent* (see Section 11.2.1). This operation differs from the verify operation, which results in a download to an unspecified recipe executor if the *namespace manager* is not able to perform the verification without help.

The *namespace manager* is responsible for converting the form of a *managed recipe* to that of an *execution recipe* for downloading. This is accomplished through the following steps:

1. If the recipe is not *verified*, then no conversion is performed. The attributes ExecAttrLength and ExecAttrChgTime are set and maintained only by the *recipe executor*. These attributes are not sent with the downloaded recipe.

2. The *generic* attributes ExtRef and Parameters of the *verified* recipe are not sent with the downloaded recipe.

3. If an *agent-specific dataset* exists that corresponds to the destination *recipe executor*, and if the *agent-specific* attributes AgentSpec Comments and Certified are non-empty, they are included in the attributes of the downloaded recipe.

4. The *execution recipe* attribute ExecLinkParam combines the contents of the *generic* attribute LinkParam and the *agent-specific* attribute AgentSpec LinkParam. If an *agent-specific dataset* exists that corresponds to the destination *recipe executor*, and if there are *variable parameter initial values* and *restrictions* in the *agent-specific* attribute AgentSpec LinkParam, then they replace their corresponding elements in the *generic attribute* LinkParam, and the results are placed in the ExecLinkParam attribute of the downloaded recipe.

The *download recipe* operation is invoked by the message service RMNAction.

9.4.9 *Upload Recipe* — The **upload recipe** operation causes the *manager* to upload a recipe from the *recipe executor* of a specified *agent* (see Section 11.2.3). This operation is the equivalent of a *namespace-initiated recipe create* (if the specified recipe *identifier* does not already exist in the *namespace*) or a recipe update.

The *upload* operation allows recipes that have been *created* or modified by the *recipe executor* to be placed under management in a *namespace*. Such recipes are always *unlinked*.

In addition, it is possible to upload previously downloaded recipes. However, due to the differences in the attributes, certain intermediate information available in a *managed recipe*, such as ExtRef and Parameters, must be re-created for the uploaded recipe by requesting the *recipe executor* to *verify* the recipe and send the results.

In the case of the *derived object form* recipe, where the original source recipe identified in the SrcRcpID attribute (common to both recipe types) exists within the *namespace*, most of the attributes of the *derived object form* recipe are identified to those of the original. The exceptions are the attributes AttrLength, AttrChgTime, BodyLength, BodyFormat, and EditTime, which are all set by the *recipe executor* at the time the *derived object form* recipe is derived.

The *namespace manager* is responsible for converting the uploaded *execution recipe* to a *managed recipe* for storage in the *namespace*. This is accomplished through the following steps:

1. If the recipe is not *verified*, then no conversion is performed. The attributes ExecAttrLength and *ExecAttrChgTime*, if uploaded, are discarded.

2. The *generic* attributes ExtRef and Parameters of the *verified* recipe are not sent with the uploaded recipe. They must be obtained separately through the *verify*.

3. If an *agent-specific dataset* exists that corresponds to the destination *recipe executor*, then if the *execution recipe* attributes AgentSpec Comments and Certified are non-empty, they are placed in the corresponding attributes of an *agent-specific dataset* for that *recipe executor*. If necessary, an *agent-specific dataset* is created.

4. Any *variable parameter initial values* and *restrictions* in the *execution recipe* attribute AgentSpec-LinkParam replace their corresponding elements in the corresponding *agent-specific attribute* (if the *dataset* for that *agent* exists) in the uploaded recipe. If it can be determined that the contents of AgentSpec LinkParam are not different from those of the *generic* attribute LinkParam (as in the case of a *derived object form* recipe), then AgentSpec LinkParam should be discarded. If necessary, an *agent-specific dataset* is created for the appropriate *agent*.

The *upload recipe* operation is invoked by the message service RMNAction.

9.5 *Synchronization* — In addition to the explicit operations that are invoked through specific message services, the *recipe namespace manager* may provide the optional capability of **sychronization** of the *managed recipes* with *execution recipes* stored by the *recipe executors* of its *member agents*. This section describes the *synchronization* capability.

The ExecChgCtrl attribute of a recipe is used to specify types of permitted changes in *execution recipes*. The *recipe executor* is permitted to change the recipe body or to save the last settings used for *variable parameters* in the ExecLinkParam attribute of the *execution recipe* only when expressly granted permission in ExecChgCtrl.

ExecChgCtrl may also require the *recipe executor* to send a **change notification** message to the recipe's **originating namespace**. The **originating namespace** is either the *namespace* from which the recipe was *downloaded* or to which a new recipe will be *uploaded*. *Change notification* applies to both the explicitly permitted changes (modification and saving the *last value*) and to a *derived object form recipe* built from a *source form recipe*. *Change notification* informs the *namespace* that a change of interest to the *namespace* has occurred.

Namespaces with *synchronization* capability provide two additional attributes, SynchOn and SynchFail. The first allows the user to disable *synchronization* or to select the types of synchronization desired, and the second records *execution recipe specifiers* of recipes for which *synchronization failed*.

Synchronization for a new or changed recipe, or a new recipe *form*, consists of *uploading* the *execution recipe* for which a *change notification* has been received and, when necessary to protect a *read-only* recipe, assigning it a new *version number* and requesting the *recipe executor* to rename the corresponding *execution recipe*. Note that the *recipe executor* is required to deny attempts to rename a currently *selected execution recipe*.

The *recipe executor* saves the *last value* of a *variable parameter* in the *execution recipe* attribute Exec-LinkParam. **Synchronization** for a new *last value* consists of getting the value of this attribute from the *recipe executor* and updating the AgentSpec LinkParam

attribute of the *agent-specific dataset* for the *recipe executor's agent*. Note that attributes of a recipe may change without affecting the *version number*.

Synchronization may fail either through failure to properly *upload* an *execution recipe*, through failure to properly retrieve the value of the *execution recipe's* ExecLinkParam attribute, or through failure to successfully rename the recipe stored by the *recipe executor*. The attribute SynchFail contains a list of *recipe specifiers* of the *execution recipes* for which *synchronization* was attempted but failed to be successfully completed. A *recipe specifier* shall be deleted from SynchFail if a later attempt at the failed operation for that recipe is successful. The *authorized user* may also remove one or all *recipe specifiers* from this attribute.

The attribute SynchOn is set by the user to indicate the types of changes for which *synchronization* shall be performed. SynchOn may be set to specify *synchronization* for changes to the body, changes to the *last value*, creation of a new recipe, building a new *derived object form* recipe from a *source form*, or any combination of these settings. A value of zero disables *synchronization*.

SynchOn is an unsigned integer. Possible values are either 0 (disabled) or any combination (sum) of one or more of the following decimal values:

0 = synchronization disabled

1 = changes in body

2 = new *execution recipes*

8 = changes in the *last value* of one or more *variable parameters* (i.e., to the ExecParam attribute of the *execution recipe*)

16 = new *derived object form execution recipes*

NOTE: Where possible, the values of SynchOn and ExecChgCtrl address the same change issues. For this reason, a value of 4 is not used, and a new value of 16 is added.

9.6 *Table of Operations* — Table 9.1 lists all the operations defined for *namespace management*.

The column labeled "Rqmt" is used to indicate those operations that are required for *fundamental compliance* to RMS as a *recipe namespace resource*.

Table 9.1 Namespace Operations

Operation	*Description*	*Rqmt*
create namespace	A new *namespace* is created and assigned an identifier.	N
delete namespace	A *namespace* is deleted.	N
rename namespace	The *namespace identifier* is re-assigned.	N
get available storage	Determine the amount of recipe storage available.	Y
check recipe status	Determine the existence and *read-only* status of a recipe, and obtain the next numeric *version*.	Y
get best version	Determine the default version for a given recipe class and *name* and for an optional *agent*.	Y
create recipe	A new recipe body is entered into the namespace.	Y
delete recipe	A recipe's identifier is removed from the *namespace.*	Y
store recipe	A recipe is stored in a namespace.	Y
retrieve recipe	A recipe is sent from the namespace.	Y
copy recipe	A new recipe is originated as a copy of an existing recipe.	N
rename recipe	A recipe is assigned a new identifier.	N
verify recipe	Check the syntax or format of a recipe body for correctness.	Y
download recipe	A recipe is downloaded to a specified agent's *recipe executor*.	Y
upload recipe	A recipe is created or updated by uploading a recipe *body* from a specified agent's *recipe executor*.	N

9.7 *Namespace Events* — A user of recipe namespace services is potentially interested in any change that occurs to or within a namespace that was not initiated by the user itself. Two such events are defined: Recipe Namespace Change and Recipe Change. A Recipe Namespace Change event occurs when a namespace is created, deleted, or renamed, or when a recipe is created, deleted, copied, or renamed. A Recipe Change event occurs whenever the body or any of the attributes, including *agent-specific attributes*, of an existing recipe is changed.

The selection of events to be reported, and the mechanisms for reporting these events, are defined in SEMI E53 (Event Reporting).

10 Distributed Recipe Namespace Management Operations

This section defines the operations required for the *distributed recipe namespace* capability. Operations are defined for the *DRNS segment*, the *DRNS recorder*, and the *DRNS manager*, in that order. Support for the *distributed recipe namespace capability* is not required for RMS compliance.

10.1 *Distributed Recipe Namespace Segment Operations* — This section defines the operations that shall be supported by the *distributed recipe namespace segment*.

10.1.1 *Object Services* — The *DRNS segment* is considered to own the recipes that it stores.

The **segment specifier** is the object specifier for a *DRNS segment* and has the form "type1:id1>...>type2:id2". An attached *DRNS segment* is owned by the *agent* providing the *DRNS segment* capabilities, by the *distributed recipe namespace* of which it is a component, and by the *DRNS manager* to which it is attached, and shall be accessible by any of these three paths. An unattached *DRNS segment* is owned by the providing *agent*.

An example of a *segment specifier* for a *DRNS segment* named ABC_Etch_Seg, a component of a *namespace* named WetEtchA provided by an *agent* named WetEtch003, is

"Agent:WetEtch003>RNSD:WetEtchA>RNSDSegment:ABC_Etch_Seg".

For a *master segment* Alpha provided by *agent* RecipeServer, this becomes

"Agent:RecipeServer>RNSD:WetEtchA>RNSDMaster:Alpha."

NOTE: The form of the specifier used for *DRNS segments* and *DRNS recorders* will vary. For example, to specify a *segment* to be attached to a *DRNS manager*, the specifier must include the object type and identifier for the *agent* providing the *segment* capability. Once the *segment* is attached, it may be specified through the *namespace* hierarchy, as in the example above.

10.1.1.1 *Attribute Read/Write* — The *DRNS segment* shall support the get attributes operation for itself and all recipes that it has stored.

It shall support the set attributes operation for its recipes only according to the restrictions against change defined in Section 5.4.2 and within Section 10.1.

If requested to change *read/write* attributes, it shall request and receive permission to change attributes prior to making such change.

A request to change either several *generic* attributes at the same time, or several *agent-specific* attributes for a specific *agent-specific dataset* at the same time, is considered for approval purposes as one change. However, changes to both *generic* and *agent-specific* attributes shall not be included in one change request or change.

10.1.1.2 *Create and Delete Operations* — The *segment* may support both the create object and delete objects pair of operations. The *authorized user* who invokes the create object operation shall assign a name to be used as its *name* ObjID. The *name* "Default" is prohibited. Once *created*, the *segment* shall be attached to a specific *manager* before it is permitted to accept recipes.

The create and delete operations are optional if the *owner agent* that provides the *distributed recipe namespace segment* capabilities provides an installed *distributed recipe namespace segment* that cannot be deleted. In this case, a means of recreating the *segment* shall be provided in the event the *segment* becomes damaged. If one of these two operations is supported, both are required.

The attributes Namespace, NamespaceManager, and RecipeReadOnlyLevel shall be set to null values at the time the *segment* is *created*.

A *segment* that is attached or that contains recipes (is not **empty**) shall not be *deleted*.

10.1.1.3 *Object Attachment Operations* — The *segment* shall support the operations to attach and reattach to a *DRNS manager*, and also the operations invoked by its *manager* to detach itself from a *DRNS namespace manager*. It shall also support the attach set attributes operation. Certain RMS operations shall be accepted only when received from its *manager*, as indicated below.

All requests for changes to recipes within the *segment* shall be sent to the *DRNS manager* to which it is attached.

The *manager* shall set the *segment* attributes Namespace, NamespaceManager, and RecipeReadOnlyLevel when attaching or reattaching a *segment*. It may change these attributes for an attached *segment* at any time.

The detach operation breaks the logical connection between the *segment* and its *manager*. The *segment* becomes unattached, and the values of Namespace, NamespaceManager, and RecipeReadOnlyLevel are set

to a null value. All recipes and recipe attributes are considered as *read-only* during the time the *segment* is unattached.

The reattach object operation is used in rebuilding a *distributed recipe namespace*. This operation sets the value of the attribute NamespaceManager to the name of the new *manager*. The *segment* returns a new token value to the *manager*.

The request to reattach itself serves to inform the *segment* that any of its *pending change requests* not previously approved have now been forgotten. The *segment* should either discard the request or resubmit it to its new *manager*.

10.1.2 *Segment Recipe Management Operations* — Within a *distributed recipe namespace* environment, all recipe management operations (Section 8) and namespace recipe operations (Section 9.4) are performed by the *DRNS segment*. The *DRNS segment* shall support all operations defined in these sections.

Operations defined in Section 9.4 that involve a *recipe executor*, such as *download* and *upload*, shall be performed only with a *recipe executor* that is either owned by the *agent* providing the *DRNS segment* capability or is owned by a component within the internal hierarchy of that *agent*. For example, a *DRNS segment* provided by a cluster tool may *download* to a *recipe executor* owned by a cluster module but not to equipment external to the cluster.

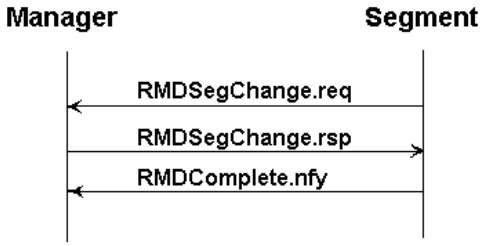


Figure 10.1

Segment-Initiated Change Request Message Flow

10.1.2.1 *Requirements for Approval* — Operations that change a recipe in any way shall be denied unless the *DRNS segment* is attached to a *DRNS manager*. All changes to logical recipes within a *DRNS segment* shall be approved by the *DRNS manager* before the changes are made to the recipe stored within the *DRNS segment*.

Changes to *agent-specific datasets* stored by attached *dedicated segments* are pre-approved. This is possible since, at most, one *dedicated segment* has an *agent-specific dataset* for a specific *agent* for any given recipe. However, the *segment* shall notify its *manager* immediately after any such change by sending the RMDNotify notification, which shall include the attribute AgentSpec_Agent, to identify the *agent-specific dataset*, as well as all *agent-specific* attributes that changed, regardless of whether they have been reset to their default value or set to a non-default value. (Otherwise, the entire *agent-specific dataset* would be required.)

A *master segment*, however, is prohibited from changing an *agent-specific dataset* without explicit permission from the *manager*.

Requests for changes that are made with RMS services defined in Section 12 may be sent from any service user, including the *DRNS manager*. The *DRNS segment* may reject requests for invalid changes, such as a request to modify a *read-only* recipe. Otherwise, the *DRNS segment* shall request approval from its *DRNS manager* for each change (Section 10.3.5). The *manager* responds by either approving, denying, or putting the request on hold.

NOTE: Service requests sent directly to a *DRNS segment* may not, in some circumstances, be fulfilled.

If the *change request* is denied, then the change is prohibited immediately. If it is put on hold, the *segment* shall retain the information necessary to effect the desired change at a later time.

If the *change request* is approved, the *DRNS segment* shall proceed with the change and shall notify its *DRNS manager* when the change is completed, either normally or abnormally, through sending the notification RMDComplete with the results, as illustrated in Figure 10.1.

The *DRNS manager* will put a *change request* on hold when another *change request* exists for the same recipe and the recipe is *locked*. In this case, the *DRNS manager* responds to the *segment's change request* with an *operation identifier* that the *manager* uses later when sending a **segment change approval** request.

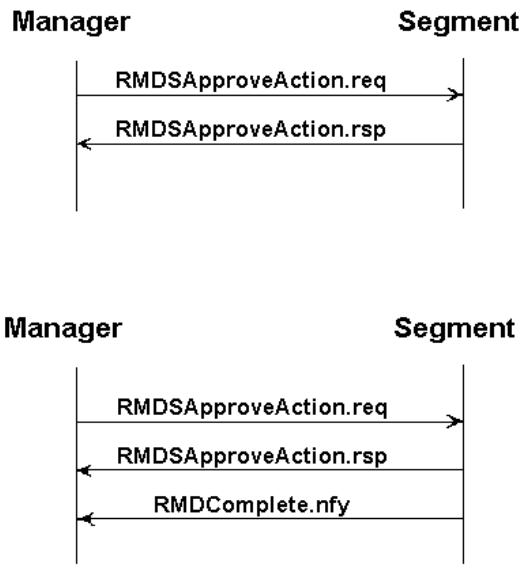


Figure 10.2

Message Flow for Manager Approval of Change Request

10.1.2.2 *Segment Change Approval* — The *DRNS segment* may receive a **segment change approval** request for an action that the *DRNS segment* had earlier requested. This informs the *DRNS segment* that an earlier *change request* made by the *DRNS segment* has now been approved for immediate action or has been completely denied.

The *segment* shall respond in one of three ways:

— It may reject the approval due to changes in circumstances since the original request was made (Figure 10.2(a).

— It may first fulfill the change request and then respond that the change request has now been completed, either successfully or unsuccessfully (Figure 10.2(a).

— It may first respond that it accepts the change request and then attempt to complete the change. When change request has been completed, normally or abnormally, the *segment* shall notify the *manager* of the results with the RMDComplete notification service (Figure 10.2(b).

Rejection of *segment change approval* shall be used when the change is no longer desired. For example, a *DRNS segment* "S" may request to *link* a *write-protected* recipe, and the *manager* responds that the request is on hold. Before the *segment* receives approval for that change, the same *logical recipe* is *linked* by a different attached *DRNS segment*, and the results are updated to each *DRNS segment* having a copy of the *logical recipe*, including *segment* "S". In this case, the recipe may not be relinked.[11]

The *segment change approval* operation is invoked by the message service RMDSApproveAction by the *segment's DRNS manager* and is otherwise invalid.

10.1.2.3 *Scenario of a Segment Change Request* — A typical scenario is illustrated in Table 10.3.

In this scenario, a local operator wants to change the generic attribute EditedBy. The *segment* requests approval for a generic attributes change. However, an *active change request* exists for this recipe.

11 Both the DRNS manager and the DRNS segment are responsible for compliance to RMS namespace requirements

Table 10.3 Typical Change Request Scenario

Manager	*Segment*
	Receives request for attribute change from local operator. Requests change from manager.
	<- RMDChangeRequest.req
Manager already has an active change request to link that recipe and answers that the request is on hold:	
RMDChangeRequest.rsp ->	
The active change request completes. The manager updates the change to the segment:	
RMNStore.req ->	
	The segment responds that it will make the change and notify the requestor when done.
	<- RMNStore.rsp
	The segment requests approval to make the second change:
	<- RMDChangeRequest.req
The manager approves the second request immediately:	
RMDChangeRequest.rsp ->	
	The segment stores the generic attributes sent by its manager.
	<- RMDComplete.nfy
	The segment notifies the requestor of the second change (in this case, its manager) that the change is complete.
	<- RMNComplete.nfy
When all other segments are similarly updated, the manager now approves the segment's earlier request to change generic attributes:	
RMDSApproveAction.req ->	
	The segment makes the approved change to EditedBy and then responds:
	<- RMDSApproveAction.rsp
	The segment sends notification to the operator making request for first change.
	change:
	<- RMNComplete.nfy
The manager requests the new set of generic attributes:	
RMNRetrieve.req ->	
	The segment sends the generic attributes to any requestor:
	<- RMNRetrieve.rsp

10.2 *Distributed Recipe Namespace Recorder* — This section defines the operations supported by the *distributed recipe namespace recorder*.

10.2.1 *Object Services* — The *recorder specifier* is the object specifier of the *recorder*. A *recorder* is owned by the *agent* providing the *DRNS recorder* capabilities. When attached to a *DRNS manager*, it is also owned by that *manager*. An example of the object specifier for a *recorder* named Recorder182 provided by *agent* RecorderServer is

"Agent:RecorderServer>RNSDRecorder:Recorder182>".

10.2.1.1 *Attribute Read/Write* — The *DRNS distributed recipe namespace recorder* shall support the get attributes operation for its attributes. It shall deny attempts to set its attributes through the set attributes service.

10.2.1.2 *Object Create and Delete Operations* — The *recorder* may support both the create object and delete object pair of operations. The *authorized user* who invokes the create object operation shall assign a name to be used as its *name* ObjID. The *name* "Default" is prohibited.

Once *created*, the *recorder* shall be attached to a specific *manager* before it is ready to accept data. Once *created*, the *recorder* shall set its attributes other than ObjID to null or empty values.

The create and delete operations are not required if the *owner agent* that provides the *distributed recipe namespace recorder* capabilities provides an installed *distributed recipe namespace recorder* that cannot be deleted. In this case, a means of recreating the *recorder* shall be provided in the event the *recorder* becomes damaged. If either of these two operations is supported, both are required.

10.2.1.3 *Object Attachment Operations* — The *recorder* shall support the operations to attach and reattach to a *DRNS manager*, and the detach and attach set attributes when invoked by the *DRNS manager* to which it is attached.

The *recorder* attributes Namespace and Namespace-Manager are set by the attach and reattach operations.

The detach operation breaks the logical connection of the *recorder* to the *namespace* and *managers* by setting the *recorder* attributes Namespace and Namespace-Manager to *null values*.

NOTE: In the event that a *distributed recipe namespace* becomes damaged, its *recorder* should be left attached so that it may later be reattached to a new *manager*.

The reattach operation is used in rebuilding a *distributed recipe namespace*.

10.2.2 *Add Segment Record* — The **add segment record** operation adds a given *segment* (its *object specifier*) to the *DRNS recorder's* internal list of *DRNS segments*. A request to *add* a *DRNS segment* that is already in the list shall be denied.

The *add segment record* operation is invoked by the service RMDRAddSegRecord.

10.2.3 *Delete Segment Record* — The **delete segment record** operation deletes a given *segment specifier* from the *DRNS recorder's* internal list of *DRNS segments*. A request to *delete* a *segment* not in the current list shall be denied.

The *delete segment record* operation is invoked by the service RMDRDelSegRecord.

10.2.4 *Add Change Request Record* — The **add change request record** operation adds a *change request record* to the *DRNS recorder*. The *DRNS recorder* keeps, at most, one *change request record* per recipe at any time. This is intended to represent a change currently approved and *active* for that recipe. If the *DRNS recorder* already has a *change request record* for the specified recipe, the information in the new *change request* replaces the previous information.

The contents of the *change request record* are defined in Section 10.3.7.4.

The *add change request record* operation is invoked by the service RMDRAddChgRecord.

10.2.5 *Delete Change Request Record* — The **delete change request record** operation removes a *change request record* for a specified recipe.

The *delete change request record* operation is invoked by the service RMDRDelChgRecord.

10.2.6 *Get Change Request Record* — The **get change request record** operation returns the current *change request record* for a specified recipe or assigned *segment*.

The *get change request record* operation is invoked by the service RMDRGetChgRecord and is available to any service user.

10.3 *Distributed Recipe Namespace Management Operations* — Operations defined in Sections 8 and 9 shall be supported by the *DRNS manager*. This section defines the additional operations provided by the *DRNS manager*. Operations defined in Sections 8 and 9.4 are delegated by the *DRNS manager* to an attached *DRNS segment*.

10.3.1 *Object Services* — The *DRNS manager* is considered to own the *DRNS segments* and any *DRNS recorder* currently attached to the *distributed recipe namespace*. In addition to the object services required in Section 9, the *DRNS manager* shall support the get type and get attribute name operations for its attached objects. The *DRNS manager* shall support object services directed to any of its attached objects and to recipes stored by specific *DRNS segments*.

The *DRNS manager* is considered to own the recipes that are owned through delegation by any of its attached *DRNS segments*. The Get Attributes and Set Attributes operations for a recipe may be directed to the *distributed recipe namespace*.

The object specifier for an object owned by a *DRNS manager* is formed by concatenating the object type and identifier for either the *manager* or the *namespace*, followed by the object type and identifier of each owned object in the ownership hierarchy. An example of the object specifier for a recipe XYZ;3 stored by *DRNS segment* ABC_Etch_Seg within the *distributed recipe namespace* WetEtch003 would be:

"RNSD:WetEtch003>RNSDSegment:ABC_Etch_Seg>MRcp:XYZ;3"

If an *agent* or a *DRNS segment* is specified, then the *DRNS manager* shall delegate the operation to the that *segment*. Otherwise, the operation shall be delegated to a *master segment*.

A request to set one or more *read-write* attributes of a recipe is treated by the *DRNS manager* as a *change request*.

10.3.2 *Delete Distributed Recipe Namespace* — A *distributed recipe namespace* with attachments may not be deleted.

The *delete distributed recipe namespace* operation is invoked by the service RMNDeleteNS defined in Section 9.2.2.

10.3.3 *Attach and Detach Supervised Objects* — This section defines the support required for the *authorized user* to request a *DRNS manager* to attach or detach one or more *segments* or a *recorder*. The operations and services are defined in detail in SEMI E39 (OSS).

10.3.3.1 *Attach Supervised Object* — The **attach supervised object** operation is invoked by an *authorized user* to request the *DRNS manager* to attach a specified *unattached segment* or *recorder*.

When a request to attach a supervised object is accepted, the *DRNS manager* sends an attach or detach request to the specified object.

The *DRNS manager* shall have the capability of managing at least one attached *dedicated* and one attached *master segment* at a time. At most, one *recorder* shall be attached to a *DRNS manager* at any given time.

Once attached, the *segment* or *recorder* becomes a formal part of the *namespace* and is owned by the *manager*.

When adding attachments to a *DRNS manager*, the *recorder* should be added first, so that it may be used to record the *segments* as they are subsequently attached. When the *distributed recipe namespace manager* receives a request to attach a *segment*, it sends that request to the specified *segment*. If the operation is successful, and if a *recorder* is attached, the *manager* requests the *distributed recipe namespace recorder* to record the *segment specifier* (the object specifier for the segment). If the *agent* providing the *segment* is not already in the *namespace* attribute Members, it is added at this time.

The operation of attaching a *recorder* shall set the *distributed recipe namespace* attribute Recorder to the value of the *recorder's* attribute ObjID. The operation of attaching a *segment* shall add the *segment specifier* to the Segments attribute of the *distributed recipe namespace*.

When a *recorder* is attached, all subsequent operations that attach and detach *segments* shall update the *recorder* through its operations to add and remove a record of the *segment*.

Specifiers used in the *namespace* attributes Recorder and Segments, and *segment* specifiers stored in the *DRNS recorder*, shall use the form including the specifier for the *agent* providing the capability for the *recorder* or *segment*. This is required for identification outside the scope of the current *namespace*. For example, if it later becomes necessary to rebuild the *distributed recipe namespace*, then the *segment* must be located through its *agent* owner rather than through the *namespace*.

10.3.3.2 *Detach Supervised Object* — An attached *segment* or *recorder* may be detached at any time. The *manager* forwards the request to the specified object. When a *segment* is detached, if a *recorder* is attached, then the *manager* requests the *recorder* to remove the *segment* that is being detached.

The user may request a *DRNS manager* to detach an attached *recorder* or *segment* at any time. When a *recorder* is detached, the *distributed recipe namespace* attribute Recorder shall be set to a null value. When a *segment* is detached, its *specifier* is removed from any attached *recorder* and from the Segments attribute of the *distributed recipe namespace*.

10.3.4 *Change Request Management* — A *change request* occurs whenever the *DRNS manager* receives any request, from any source, to change a recipe or the contents of the *distributed recipe namespace* as a whole. This includes requests to change the recipe *identifier*, a *generic attribute*, an *agent-specific attribute*, or the body of an existing recipe. It also includes all changes that affect the set of *recipe identifiers* within the *distributed recipe namespace*.

Requests for changes may come from a source that is either internal or external to the *namespace*.

10.3.4.1 *External Change Requests* — The *DRNS manager* may receive a request, through recipe