



SEMI E42-95

RECIPE MANAGEMENT STANDARD: CONCEPTS, BEHAVIOR, AND MESSAGE SERVICES

1 Introduction

This provisional standard defines the concepts required for management of recipes, the operations or behavior provided by the Recipe Management Standard (RMS), and the messages through which services are provided through an interface between the provider and the user of these services.

1.1 Purpose — The purpose of this standard is two-fold:

- To enable applications software to be developed that can assume the existence of standard concepts, behaviors, and message services that collectively form Recipe Management and that take advantage of them.
- To enable software to be developed to offer the Recipe Management capabilities.

1.2 Scope — This is a provisional standard that defines concepts, behavior, and services to support the integration of automated recipe management within a semiconductor factory. These services are applicable to a variety of relationships, including both traditional **host/equipment** and cluster tool controller/attached module communications and control.

The standard provides a set of communications services which allow such systems to transfer and manage recipes to ensure the correct processing of material within semiconductor manufacturing *equipment* and systems. RMS also requires compliance to SEMI E39 (Object Services Standard (OSS): Concepts, Behavior, and Services) for completeness.

The provisional status of RMS will be removed after the following sections have been added:

- **execution recipe** concepts, Section 3.5,
- **distributed namespace** concepts, behavior, and services, Sections 5, 10, and 13,
- **recipe executor** concepts, behavior, and services, Sections 6, 11, and 14,
- RMS Compliance, Section 15, a collection of requirements defined in preceding sections, and
- Glossary of RMS terminology, Section 16.

The models, definitions, and services for the distributed recipe namespace are not required for implementation of basic recipe namespace capability. However, they will extend the capability of recipe namespace through the management of distributed storage. Also, the section on compliance will organize the various requirements into tables for clarification.

This document describes several different hierarchical relationships: supervisory agents and their supervised agents, recipes and their subrecipes, and recipe classes and their subclasses. Such hierarchical relationships provide a natural organizational and classification structure that is reflected in many different kinds of systems, such as telephone switching systems and directory trees. It is the intent of this standard to support logical hierarchical relationships rather than to require that strict hierarchical relationships be implemented in systems architecture.

RMS places no restriction on where the set of defined services is implemented.

1.3 Referenced Documents

1.3.1 Semiconductor Equipment and Materials International (SEMI)¹

1.3.1.1 SEMI Equipment Automation/Hardware Volume

SEMI E10 — Guidelines for Definitions and Measurements of Equipment Reliability, Availability, and Maintainability (RAM)

1.3.1.2 SEMI Equipment Automation/Software Volume

SEMI E30 — Generic Model for Communications and Control of SEMI Equipment (GEM)

SEMI E39 — Object Services Standard: Concepts, Behavior, and Services

Draft Document 2309 — Event Reporting Standard (ERS)

1.3.2 Other References

¹ Semiconductor Equipment and Materials International (SEMI), 805 East Middlefield Road, Mountain View, CA 94043, (415)964-5111, FAX (415) 967-5375



James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987.

1.4 Definitions — Basic definitions for objects, services, and form are provided in SEMI E39, Sections 4.1 through 4.4. This section provides additional definitions.

Definitions in Section 1.4.4 are specific to RMS.

1.4.1 Objects

standardized object — a object formally defined in SEMI standards and in compliance with the fundamental requirements of SEMI E39 (Object Services Standard: Concepts, Behavior, and Services).

1.4.2 Services

service provider — an application (a component of an *agent*) responsible for providing services to the service user.

service user — (service consumer) an application that uses the services provided.

1.4.3 Form

binary — a string of bit values (zeroes and ones), with a format that is either left unspecified or specified by bit position, with the most significant bit first. The total length of the string is a multiple of eight. Messaging protocol may impose restrictions on length.

1.4.4 Recipe Management

This section introduces basic terminology used in RMS. Entries are in alphabetical order. Additional definitions and specifications are provided in later sections.

agent — an intelligent system within a factory that provides one or more service resources and uses the services of other agents. This is a generalization that includes host, equipment, cell, cluster, cluster module, station controller, and work station. *Agents* are associated with a physical system or a collection of physical systems, such as computer platforms.

authorized user — a user who can be identified to an *agent* as having the level of authority required for a

particular activity, such as *certifying* a recipe for that equipment.

collection event — A detectable occurrence of interest to a service user.

component agent — a subordinate *agent* that provides services to a *supervisory agent*.

download — an operation that transfers a recipe (down) to an execution storage area.

edit — an operation which creates a new recipe body or changes the body of an existing recipe.

editor — a service which allows a *user* to edit a recipe. *Editors* are not specified in RMS.

equipment² — An *agent* with associated hardware that provides, at a minimum, recipe execution services.

event — A detectable occurrence significant to an object.

execution (recipe execution) — the process of reading the recipe contents and implementing its instructions, process parameters, or other information required for its own processing.

executing agent — an *agent* that provides *recipe execution* capabilities.

execution area — the storage location of the recipe(s) currently *selected* (ready) for *execution*.

host — a *supervisory agent* that represents the factory to its subordinates.

logical recipe — a recipe with a particular set of *attributes* and a particular *body*, considered independently from its physical location. A *logical recipe* may have multiple instances or copies.

name — a text-based *attribute* of an object that may be used as all or part of its *identifier*.

namespace — in general, a domain within which object identifiers are unique. In RMS, the term *namespace* is used as a synonym for *recipe namespace*, unless otherwise stated.

operator — the user who interacts locally with *agent* through the *agent's* interface.

recipe — the pre-planned and reusable portion of the set of instructions, *settings*, and parameters under con-

² The term "equipment" is restricted in RMS to "intelligent equipment."



trol of an *agent* that determines the processing environment seen by the manufactured object and that may be subject to change between runs or processing cycles.

recipe class — a formal grouping of recipes with a common language syntax and functionality.

recipe executor — the component of an *executing agent* that executes recipes.

recipe namespace — a logical management domain with the responsibility for the storage and management of recipes, the ensurance of the uniqueness of recipe **identifiers** within that domain, and the provision of services pertaining to recipes stored within that domain.

recipe parameter — a control value that affects the *agent's* process.

select — the act of preparing a recipe for execution.

setting — a static value accessible to the *user*, through one or more methods, that is used by equipment to control its process. *Settings* include, but are not limited to, setpoint values. *Settings* typically may be specified within a recipe.

storage area — an area where objects and data are stored.

subordinate agent — an *agent* that is a component of, or managed by, another *agent*.

supervisory agent — an *agent* with supervisory responsibilities for one or more subordinate *agents*.

timestamp — the notation of the date and time of the occurrence of an event.

upload — an operation that transfers a recipe (up) from an execution storage area.

user — a person interacting with an *agent*, directly through the *agent's* human interface or indirectly through the *agent's supervisor*.

validate — the action of checking recipe contents to ensure that parameter type and range are valid for the equipment configuration prior to execution. [Note that *validation* and *verification* are used in different ways.]

variable parameter — a formally defined variable (*setting*) defined in the body of a recipe, permitting the actual value to be supplied externally.

verify — the operation of reading a recipe's contents to ensure that it is syntactically correct and identifying elements that must be made public.

version — part of a recipe's *identifier* that is used to show its heritage.

1.5 Conventions

The following conventions are used in this document:

- To highlight terms specific to RMS (excluding terms defined in Sections 1.4.1 through 1.4.4 that are common to multiple standards), a defined term appears in **boldface** wherever it first appears and wherever it is defined. This alerts the reader to those terms with specific meanings. Except for terms that are very common, such as recipe, host, and equipment, defined terms otherwise are in *italics* wherever they appear.
- Terms related to objects and object services are in conformance with SEMI E39 (Object Services Standard: Concepts, Behavior, and Services). A brief discussion of objects is provided in the Appendix of that document.
- Attribute names are underlined.
- Attributes called "name" in RMS are generally intended to be used for the ObjID attribute of a *standardized object*.
- To prevent the definition of numerous message parameters named "XxxList," this document adopts the convention of referring to the list as "(List of Xxx)". In this case, the definition of the parameter Xxx will be given, not of the list. The term "list" indicates a collection (or set) of zero or more items of the same data type.

For attributes that are lists, this convention is not followed, as the entire attribute, as a list, must be assigned a specific name.

1.5.1 Text String Restrictions — Text strings used in attribute names, attribute values, or message parameters, are subject to the restrictions defined by OSS: Text in ASCII is restricted to the characters between 20₁₆ and 7D₁₆, excluding the question mark "?" (3F₁₆), the asterisk "*" (2A₁₆), and the tilde "~" (7E₁₆).

Text strings used as, or within, the object identifier ObjID are additionally restricted to exclude the "greater than" symbol ">" (2E₁₆) and the colon character ":" (3A₁₆) to conform with OSS requirements.



1.5.2 Harel State Model — This document uses the Harel State Chart notation to describe the dynamic behavior of the objects defined. An overview of this notation is presented in an Appendix of SEMI E30. The formal definition of this notation is presented in Science of Computer Programming 8, "Statecharts: A Visual Formalism for Complex Systems," by D. Harel, 1987.

This document also adopts the extension of Harel notation to show the deletion of an object as used by Rumbaugh, et al. (Section 1.5.2.1).

1.5.3 Objects — Standardized objects defined by RMS conform to the requirements of SEMI E39 (Object Services Standard: Concepts, Behavior, and Services). RMS adopts the convention of showing *standardized objects* as drawn with a heavy line in object models. *Non-standardized objects* are used to illustrate concepts and relationships, but are not formally defined and cannot be accessed through Object Services. A list of *standardized objects* defined by RMS is included in Related Information 1.

1.5.3.1 OMT Object Information Model — The object models are presented using the Object Modeling Technique, developed by Rumbaugh et al., in *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991. Overviews of this notation are provided in an appendix of SEMI E39 (Object Services Standard: Concepts, Behavior, and Services).

1.5.3.2 Object Attribute Representation — The object information models for *standardized objects* will be supported by an **attribute definition** table with the following column headings:

Attribute Name	Definition	Access	Rqmt	Form
The formal text name of the attribute.	Description of the information contained.	RO or RW	Y or N	(see below)

The Access column uses RO (Read Only) or RW (Read and Write) to indicate the access that users of the service have to the attribute.

A 'Y' or 'N' in the Requirement (Rqmt) column indicates if this attribute must be supported in order to meet fundamental compliance for the service.

The Form column is used to indicate the format of the attribute. (See Section 1.4.3 for definitions.)

1.5.4 Service Message Representation

Service Resource Definition

A **service definition table** defines the specific set of messages for a given *service resource*, as shown in the following table:

Message Service Name	Type	Description
Message name	N or R	The intent of the service.

Type can be either N = Notification or R = Request.

Notification type messages are initiated by the service provider and the provider does not expect to get a response from the user.

Request messages are initiated by a service user. Request messages ask for data or an activity from the provider. Request messages expect a specific response message (no presumption on the message content).

Service Parameter Dictionary

A **service parameter dictionary** table defines the parameters for one or more services, as shown in the following table:

Parameter	Form	Description
Parameter X	Data type	A parameter called X is B in A.

A row is provided in the table for each parameter of the service. The first column contains the name of the parameter. This is followed by columns describing the form and contents of the corresponding primitive.

The Form column is used to indicate the type of data contained in a parameter. (See Section 1.4.3 for definitions.)

The Description column in the Service Parameter Dictionary table describes the meaning of the parameter, the values it can take on, and any interrelationships with other parameters.

To prevent the definition of numerous parameters named "XxxList", this document adopts the convention of referring to the list as "(List of)Xxx". In this case, the definition of the variable Xxx will be given, not of the list. The term "list" indicates a collection (or set) of zero or more items of the same data type. Where a list is used in both the request and the response, the list order in the request is retained in the response. A list



must contain at least one element unless zero elements are specifically allowed.

Service Message Definition

A **service message definition** table defines the parameters used in a service, as shown in the following table:

<i>Parameter</i>	<i>Req/Ind</i>	<i>Rsp/Conf</i>	<i>Description</i>
Parameter X	(see below)	(see below)	A description of the service.

The columns labeled Req/Ind and Rsp/Conf link the parameters to the direction of the message. The message sent by the initiator is called the "Request". When the receiver receives this message the "Indication" or the request. The receiver may then send a "Response", which the original sender terms the "Confirmation".

The following codes appear in the Req/Ind and Rsp/Conf columns and are used in the definition of the parameters (e.g., how each parameter is used in each direction):

"M" — **Mandatory parameter** — must be given a valid value.

"C" — **Conditional parameter** — may be defined in some circumstances and undefined in others. Whether a value is given may be completely optional or may depend on the value of the other parameter.

"U" — **User-defined parameter**.

"-" — The parameter is not used.

"=" — (for Response only) Indicates that the value of this parameter in the response must match that in the primary (if defined).

1.6 Requirements

Requirements for recipe management are varied. For example, there is a fundamental need to retain information about each recipe, to provide that information upon request, and to transfer it with the recipe so that it remains available. There is a need to classify recipes, to reuse them and to share them with multiple installations of *equipment*. The need for sharing recipes introduces a new requirement for safeguarding the relationship between a recipe's **identifier** and its contents, to be able to protect a recipe from unauthorized changes, and to ensure that the *identifier* for a recipe that is used by multiple *equipment* is in fact the same recipe.

The requirements for managing recipes in a factory are given in tabular form in Table 1.1, which presents them according to their functional areas and the specific issues involved. Table R1-1, in Related Information, shows the specific concepts defined in RMS to address each of these requirements.



Table 1.1 Requirements

<i>Functional Areas</i>	<i>Issue</i>	<i>Requirements</i>
Management of recipes	Identification	Uniquely store, identify, and select recipes in a system.
		Easy and clear identification of a recipe.
	Multiple access	Share recipes among <i>equipment</i> .
		Enable synchronized change of shared recipes among <i>equipment</i> .
	History and traceability	Capture the history of recipe usage.
		Capture the history of recipe changes.
	Life cycle management	Manage approval of recipes for process development and production.
Operations	Protection	Protect recipes from unexpected changes.
		Protect recipes from mistakes in operations.
	Portability	Create, edit, and change recipes outside the <i>equipment</i> which execute them.
	Reusability	Change recipes for a specific piece of <i>equipment</i> .
		Use recipes developed on one piece of <i>equipment</i> for other <i>equipment</i> .
Execution	Safety	Share recipes among the same kind of <i>equipment</i> by adjusting for individual differences.
		Do not execute recipes that are not syntactically correct.
		Execute the specified (selected) recipes without errors.
	Flexibility	Protect recipes being executed from inadvertent change caused by other activities.
System Operation	System Node Operation	Change a recipe's parameters during or between runs in a systematic way.
		Dynamically connect and disconnect <i>equipment</i> to/from the communications network.
	Stand-alone operation	Execute recipes with no communications link.
		Create and change recipes with no communications link.
		Manage recipes moved between <i>equipment</i> or from off-line storage through removable media.
		Enable smooth integration of stand-alone <i>equipment</i> into on-line factory systems.



1.7 Document Structure

Figure 1.1 depicts the domain of the Recipe Management Standard as consisting of Purpose, Concepts, Behavior, and Message Services. This also reflects an underlying structure of the document.

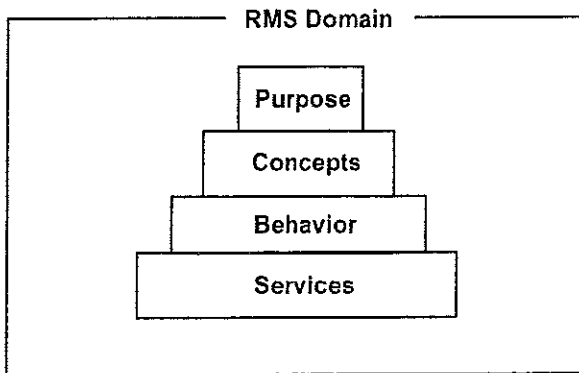


Figure 1.1
Recipe Management Domain

Purpose provides the motivation for Recipe Management capabilities and is addressed in Section 1.6. Concepts provide a detailed introduction to the *standardized objects* of RMS, their attributes, and their relationships with other objects. Behavior describes the operations that are performed by, or on, these objects. Finally, message services define the messages and their parameters independently of the protocol in which they are implemented. Both concepts and behavior represent an "inside view" of an RMS application, while services provide an interface from an external view.

- Section: 1: the formal introduction to RMS.
- Section 2: an overview of the major objects of RMS and their relationships, to provide a general context for the technical detail that follows.
- Sections 3-6: concepts for recipes, *recipe namespace*, *distributed recipe namespace*, and *recipe executor*.
- Section 7: the concept of the *agent* and of *service resources*, to provide a more complete context for RMS implementations.
- Sections 8-11: behavior (operations) for **recipe management**, **namespace management**, **distributed namespace management**, and the *recipe executor*.

- Sections 12-14: definitions of message services for the *recipe namespace*, *distributed namespace*, and *recipe executor service resources*.
- Section 15: RMS compliance.
- Section 16: a glossary of terms, provided as a convenient reference.
- Related Information: provides background information on the requirements behind RMS and examples of applications.

1.8 Applicable Documents

ISO/TR 8509:1987, Information Processing Systems, Open Systems Interconnection — Service Conventions.

2 Overview of RMS

This section provides an introduction and overview to the major objects of RMS.

2.1 Recipe Management Models — To provide a clearer understanding of the major entities or areas of functionality specified by RMS, they are portrayed as objects in RMS models, using OMT notation.

Relationship lines in the object-based models do not represent a direct communication link between objects, but rather the knowledge and association that one object has with respect to another.

The models do not indicate the relative location of two objects.

2.2 Major Objects in RMS — Recipe Management is concerned with the major areas of functionality illustrated in Figure 2.1: recipes³, **recipe namespaces** providing persistent recipe storage, **recipe namespace managers** that provide management of the *namespace* and access to its recipes, and the **recipe executor** that executes recipes.

³ The recipe object in Figure 2.1 is not itself a standardized object. However, two recipe subtypes defined in Section 3 are standardized objects.

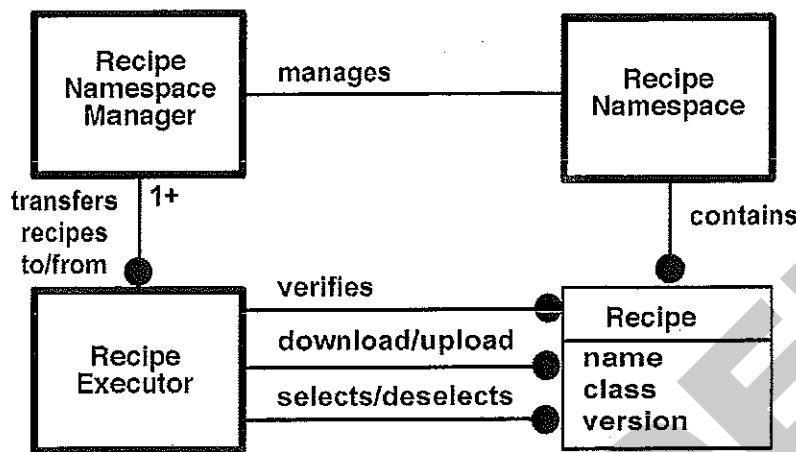


Figure 2.1
Major Objects of RMS

2.2.1 Recipes — Recipes provide a flexible, manipulatable, and re-usable form for users to select sequencing and settings to effect a particular result. Equipment uses recipes in a variety of ways to control the processing environment, sometimes using several types of recipes together. Recipes may also be used for maintenance activities, such as calibration or cleaning.

Recipes can be modified and copied from one environment to another. Their inherent flexibility, as a form, also becomes an endless source of problems unless they can be managed. Misprocessing, unintentionally running the wrong recipe for a given product, is prohibitively expensive for factories.

RMS defines a recipe as an object with attributes — information about the recipe — as well as content. It is the attributes of the recipe that allow true management of recipes, throughout their lifecycles, to occur.

2.2.2 Recipe Namespace — A *recipe namespace* in RMS is analogous to a smart file directory. The *namespace* provides long-term recipe storage capability.

Basic operations are part of the *namespace* specification. For example, recipes can be copied from one *namespace* to another, copied within a *namespace*, renamed, created, updated, and deleted. They can be downloaded to, and uploaded from, an application called a *recipe executor*. Several recipes can be *linked* together to form a set.

The *namespace* also can serve as a recipe pool that can be shared by a group of equipment of a common type. The *distributed namespace* is a *namespace* able to utilize and manage recipe storage provided by such an equipment group.

2.2.3 Recipe Namespace Manager — The *recipe namespace* itself is passive, a container for recipes. The *recipe namespace manager* (*manager*) provides the dynamic element that manages the *namespace*. The *manager* represents the interface for the *namespace* to the external world and the internal decision authority within the *namespace*.

2.2.4 Recipe Executor — The *recipe executor* (*executor*) is the component of an *agent* that understands the contents of a recipe and is able to *verify* their syntactical correctness and to *validate* that it can be executed under the current configuration. It *executes* a recipe by reading its contents and applying them appropriately in order to achieve a desired result.

The *recipe executor* also may provide additional limited storage capability, but it is restricted from modifying existing recipes, except under special conditions, to prevent unexpected and/or unwanted change.

2.3 Agent — As used in RMS, an *agent* is a system, or subsystem, in a factory, that has a physical aspect. It consists of one or more applications that provide and/or use *service resources*, such as a *recipe namespace manager* and/or a *recipe executor*, as illustrated in Figure 2.2.

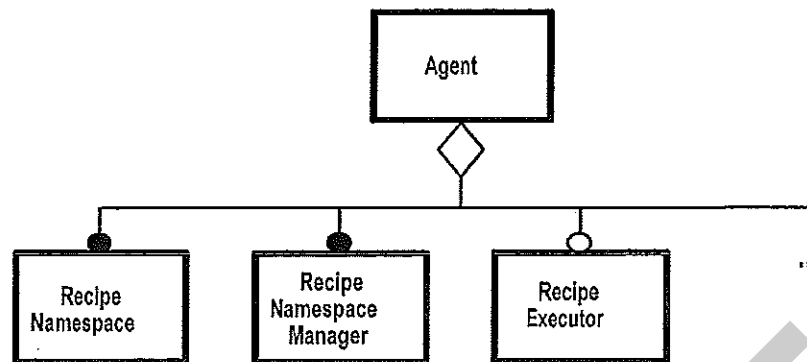


Figure 2.2

An Example of an Agent and Component Applications

Agent is a generalization that covers traditional *equipment*, supervisory or aggregate systems such as stations, cells, and clusters, and intelligent subsystems within *equipment*, such as process modules within a cluster. Other types of *agents* may also be implemented, such as those dedicated to *recipe namespace management*.

An *agent* that provides a *recipe executor* is called an **executing agent**. **Equipment** is an *executing agent* with associated hardware to which the recipe applies and which it uses to do work.

2.4 Implementations — RMS can be implemented on different platforms within a factory. A host controller may only provide *recipe namespace management*, while a diskless cluster module may provide only *recipe execution*. Traditional *equipment* capable of operating in stand-alone mode is required to provide a *recipe namespace* that is available on powerup as well as a *recipe executor* component. Additional examples of implementations of RMS are provided in Related Information 1.

RMS defines the services provided through an external interface. Communications between or within applications that are internal to an *agent* are not covered by RMS, so long as the attributes and operations comply with RMS requirements.

3 Recipes

This section describes the concepts concerning recipes.

3.1 Motivations — Definitions of recipe structure, attributes, and operations address the following issues:

- maintenance of recipe attributes for management and traceability,
- provision of a standard method for transferring attributes separately as well as with the recipe body,
- reduction or elimination of the need for dedicated recipe editors,
- support for applications requiring multiple recipe types, such as formats or language syntaxes,
- run-to-run control,
- support for sorting by factory approval and certification levels,
- the ability to tune a generic recipe for a specific installation of agent as well as the same agent over time, and
- the ability to share recipes across *executing agents* supporting the same recipe syntax or language(s).

3.2 Basic Concepts — There are two types of recipes addressed by RMS, the **managed recipe** that is stored in a *recipe namespace* and the **execution recipe** that is stored in **recipe execution storage**. This section describes the basic concepts concerning both types of recipes.

3.2.1 Types of Recipes — There are three types of recipe objects shown in Figure 3.1. Most of Recipe Management is concerned with the management of recipes stored in a *recipe namespace*. These recipes are called **managed recipes**. Recipes are also stored by the *recipe executor* in the **recipe execution storage**, and these are called **execution recipes**.

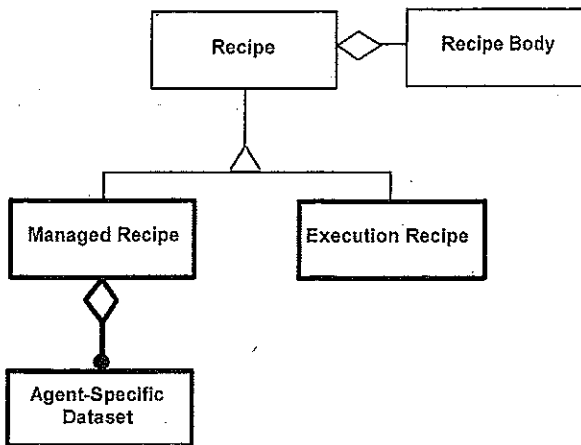


Figure 3.1
Recipe Types

The recipe supertype is an abstract type that shows the common elements and attributes of the *managed recipe* and the *execution recipe* and their differences. The recipe supertype is not itself intended to be implemented. That is, recipe object types formally defined in RMS are either *managed recipes* or *execution recipes*. As indicated by the light lines, the supertype is not a *standardized object*.

3.2.2 Recipe Structure — Both the *managed* and the *execution* recipe have a **body**⁴ and a set of attributes. The *body* (contents) of the recipe contains the data used by the *recipe executor* for its execution process. The *body* is also not a *standardized object* and may not be accessed through Object Services.

Figure 3.2 illustrates all of the attributes defined by RMS that are common to both the *managed recipe* and the *execution recipe*. Not all of these attributes are required for a minimum implementation of RMS. The rules for setting the values of certain attributes, however, are different for the *managed recipe* and the *execution recipe*.

⁴ The recipe's body corresponds to a "process program" in SEMI E5 and SEMI E30.

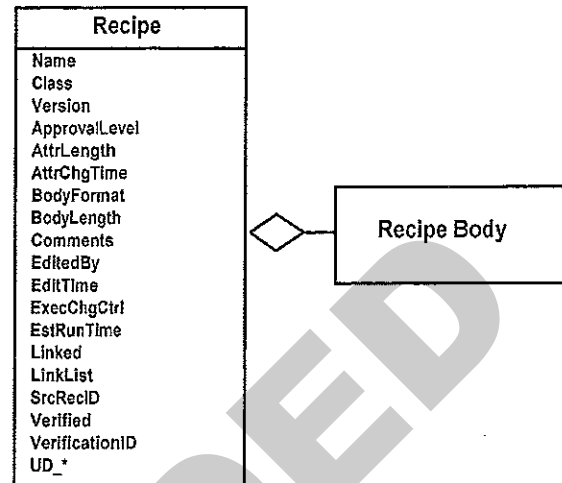


Figure 3.2
Recipe Supertype

The **identification attributes**, those attributes used for purposes of identification, are discussed in Section 3.3. The other attributes are discussed in later sections specific to each recipe type.

The *managed recipe* also may maintain a set of attributes that are specific to a single *executing agent*. The **agent-specific** attributes are technically attributes of the association between the recipe and the *executing agent*. In RMS, however, these attributes are treated as attributes of a component of the recipe called the **agent-specific dataset**, in order to allow them to be managed within the *namespace* along with the other attributes of the recipe. *Agent-specific* attributes allow the recipe to be personalized for particular *executing agents* (i.e., particular *recipe executors*).

Because the *managed recipe* may be applied to multiple *agents*, it may have multiple *agent-specific datasets*. When a *recipe* is downloaded from a *recipe namespace* to a *recipe executor*, the *managed recipe* is transformed into an *execution recipe* by taking the *agent-specific* attributes of the *agent-specific dataset* for that *agent* (its *recipe executor*) are merged into the attributes of the *execution recipe*.

Methods of storing a recipe or the components of a recipe are not dictated by RMS. However, the association between a recipe's *body* and its attributes shall be carefully maintained.

3.2.2.1 Recipe Body — The recipe **body** contains the reusable instructions, *settings*, *parameters*, and other data that the *recipe executor* reads to *execute* the recipe and control its operation.



A recipe *body* may be in one of two basic forms: **source** (text) or **object**. The purpose of the *source form* is to encourage the use of human-readable and human-editable text. The *object form* supports the types intended only to be read and applied by machines and other automated systems, such as those produced by vision systems or CAD programs, as well as proprietary formats created and used by the *recipe executor*.

3.2.2.1.1 Source Form — Recipes that are created and modified by a form of editor should be available in *source form*. The **source form** of a recipe *body* is equivalent to a text file, such as ASCII or JIS-8, that can be created, read, printed, or modified with any "standard" text editor⁵. The *executing agent* is not required to supply such a text editor, but its *recipe executor* should be able both to read and write recipes in *source form*. Editing is not covered by RMS.

The *source form* of a recipe may be copied from one *namespace* to another *namespace* and from a *namespace* to a *recipe executor's recipe execution storage*, and all *recipe executors* with access to a given *namespace* are assumed to use the same recipe language and the same functionality.

Two restrictions are placed on the interpretation of the text transferred between agents:

- In ASCII, the valid characters are from space (20₁₆) through "~" (7D₁₆), plus the the tab (09₁₆), linefeed (0A₁₆), carriage return (0D₁₆), and form feed (0C₁₆) characters. The space and tab are considered as **horizontal whitespace** characters, and the carriage return and form feed characters are considered **vertical whitespace** characters. The line-feed character is used to define the end of a line of text (eol).
- The meaning of a recipe should not depend on embedded control characters, or specific *whitespace* characters, or the existence of *whitespace* at the end of a line. *Horizontal whitespace* characters should be treated alike for purposes of interpretation, as should *vertical whitespace* characters. All other characters, aside from those specified here, are discouraged and, if

found, should be removed from the text prior to transfer.

In general, the recipe language and syntax used within a recipe *body* are beyond the scope of recipe management. However, to reduce arbitrary variability for the *user*, the following "rules" are recommended:

- Case is not significant for the purpose of comparison or meaning but should be retained as encountered, since it is often used to enhance readability. For example, the tokens `alarmlevel`, `ALARM-LEVEL`, and `AlarmLevel` should all have the same meaning.
- Comments may be included in the text and are ignored in recipe interpretation. Two types of comments are defined: comments that begin with the character pair `"/"` continue to the end of the line, while comments that begin with the character pair `"/*"` are terminated only by the character pair `"*/"`. A comment of the second type is terminated by the first character pair `"*/"` that follows. Comments may not be nested. Nested comments are not honored and may generate a syntax error.
- The minimum line consists of a single line-feed character.

In addition, recipe languages that support **external references** to other recipes should use the text form defined for the recipe *identifier* (Section 3.2.3.4) and for the recipe *specifier* (Section 3.2.4.1.3).

The supplier of the *recipe executor* shall provide documentation that formally defines each recipe language that the *agent* supports. Two forms are required, one for the *user*, who must be able to write correct recipes, and one that covers both the syntactical and lexical structures of the language using a formal descriptive protocol such as the Backus-Naur Form (BNF). The second form of documentation allows parsers to be built that are capable of pre-checking a recipe for syntactic and lexical correctness prior to downloading it. Pre-checking improves efficiency and performance by reducing the demand for recipe checking by the *recipe executor*.

The final responsibility for determining the correctness of a recipe shall belong to the *recipe executor*.

3.2.2.1.2 Object Form — Recipes that can not be translated into a meaningful *source form* include datasets derived from vision systems or mechanical systems through a special hardware-dependent *recipe*

⁵ The intent is not to specify the text editor, but rather to allow a user to modify text on the basis of clear guidelines and restrictions that may apply. Source form recipes are highly desirable to reduce the proliferation of, and need for, proprietary editors that are dedicated and language-specific.



operation. Such recipes may remain in a proprietary *object form* and may or may not be applicable to other *recipe executors* because of their hardware-dependencies.

For greater efficiency, *recipe executors* may rewrite (e.g., *tokenize*) a *source* recipe to generate a proprietary *object form* of the recipe. This is called the **derived object form**, to distinguish it from *object form* recipes that never exist in *source form*. The *agent's* manufacturer may choose to make this form available for transfer as well. From the point of view of the *supervisor*, the *body* of a recipe that is in *object form* is an unstructured binary vector or string. In general, it cannot be edited on a *supervisor*, and the *source form* of the recipe is still required. *Derived object form* recipes that cannot be used by multiple *recipe executors* should not be stored in *recipe namespaces* that are shared.

Where both the *source form* and the *derived object form* exist, the *identifier* for the *object form* of the recipe shall be different from that of the *source form* in some recognizable way. The supplier of the *recipe executor* shall provide a documented method for distinguishing one from the other within the *identifier*. This method must be such that the relationship of the *derived object form* recipe from the original *source* recipe is obvious to the user. (See Section 3.2.3.4 for examples.) An attribute of the recipe, SrcRecID, is provided to retain the relationship between the *source form* recipe and the *derived object form* recipe.

3.2.3 Recipe Identification — The ability to properly and unambiguously identify a recipe is critical to both recipe management and processing. It is also important for the user that identification be logical. RMS defines three elements used to uniquely identify a recipe within any *recipe namespace* and within the storage of the *recipe executor*. These three elements are the recipe's **class**, **name**, and version. Each element is expressed as a text string, and they are concatenated together to form the **recipe identifier** (Section 3.2.3.4).

3.2.3.1 Recipe Name — The **recipe name** is a user-defined text string that may be used to encode the technology, layer, manufacturing area, and other characteristics. The *recipe name* alone is not necessarily unique within either a given class or a given *namespace*.

The *recipe name* is subject to the conventions for text as defined in Section 1.5.1.

3.2.3.2 Recipe Class — A **recipe class** is a formal grouping of recipes that have a common syntax. A class may contain **subclasses** where the recipes within the *subclasses* operate in different environments or have different syntaxes from the parent *class* and/or from one another. A *subclass* is itself a *class* and may contain further *subclasses*.

The overall model for recipes is a set of *class* hierarchies or "*class trees*" defined by the supplier of the *recipe executor*. The major *classes* of recipes are the **PROCESS class**, the **SERVICE class**, and possibly other *agent-specific classes*. These are called **primary classes**. A **primary class** is a *class* that is not a *subclass* of another *class*.

The **PROCESS class** is a required *primary class* and is composed of recipes whose primary purpose is to increase the manufactured value of the production material⁶.

The **SERVICE class** is an optional *primary class* used for recipes whose purpose is to maintain, prepare, calibrate, or test the operation of *equipment*. Typical *subclasses* of the **SERVICE class** might be **CALIBRATION** and **CLEANING**. **SERVICE** recipes that use the same recipe language as normal process recipes are not required to be placed into a separate *primary class*. The **SERVICE class** allows separation and use of a different syntax for special non-process purposes.

Other *primary classes* that do not fit the **PROCESS** or **SERVICE** may be provided by the supplier of the *recipe executor*. Such additional *classes*, and their function, shall be documented.

Recipes that are not in the **PROCESS class** or one of its *subclasses* shall not be used for production purposes. However, recipes within the **PROCESS class** may be used for purposes other than production.

Figures 3.3 and 3.4 show examples of object models of possible *classes* and *subclasses* for a wire-bonder and for a furnace with an automatic boat-loader subsystem.

⁶ Material handling systems, wafer metrology, wafer measurement, wafer inspection systems, as well as process equipment which directly changes the characteristics of the material, increase the value of the material in some way.

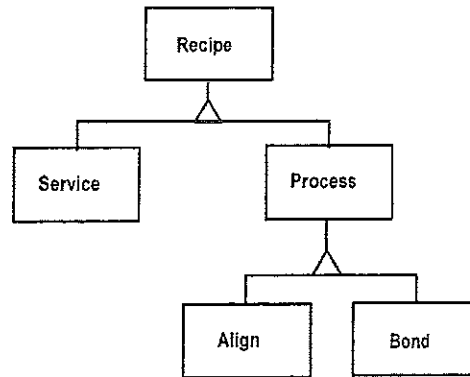


Figure 3.3
Wire-Bonder Recipe Classes

The text form of a recipe *class* shall conform to the convention for text (Section 1.5.1). Within the text form of a recipe *identifier*, a class is delimited on both ends by the forward slash character "/". The complete *class* specification within recipe *identifier* is formed by the concatenation of *classes* and *subclasses*, starting with the *primary class* and ending with the particular *class* of the recipe in question:

"/PRIMARY CLASS/SUBCLASS₁/SUBCLASS₂/...".

For example, a recipe named DryOx, version 4, within the NORMAL CYCLE *class* in Figure 3.4., has an *identifier* of

"/PROCESS/FURNACE/DIFFUSION/NORMAL CYCLE/DryOx;4".

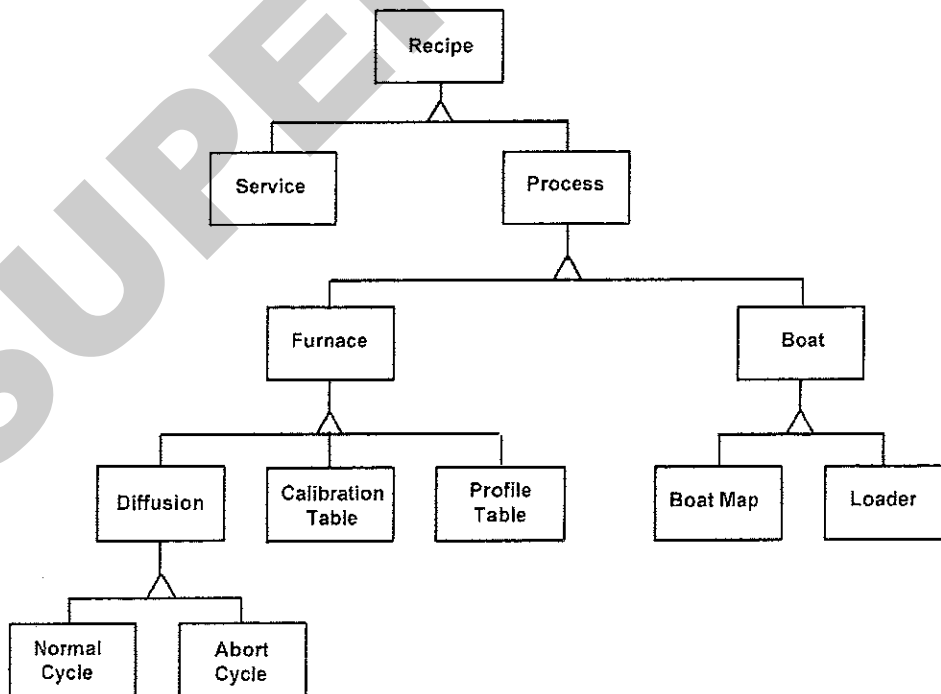


Figure 3.4
Recipe Classes for Furnace



The complete *class* specification, which always starts with the *primary class*, shall always be accepted. If *class names* within a *namespace* are all unique, however, the *service* provider may also accept a *class* designator consisting of a single *class*. The *identifier* in the example above then becomes

"/NORMAL CYCLE/DryOx;4".

3.2.3.3 Version — A recipe may evolve over time and exist in several versions. This allows the user to retain a recipe *name* over multiple versions and show the recipe's heritage. Different versions are identified by the **version** portion of the recipe's *identifier*. NOTE: More than one version of a recipe may be in use for production at the same time, not only within the factory but also within a single equipment.

A **version** is a text string consisting of at least one character. A new *version* is either generated automatically by the *recipe namespace* or is assigned by the *user*. A *user* may assign any combination of text characters and punctuation marks to the *version* except for

- the characters prohibited by the convention for text usage and the object identifier (Section 1.5.1),
- *whitespace* characters.

It is recommended, but not required, that only upper case be used.

Numeric versions (version numbers) consist only of the digits "0" through "9" and one decimal point character "." and can be translated to a pure number. To avoid confusion and multiple *versions* with the same numeric value, *numeric versions* are further restricted as follows:

- whole numbers (with no decimal point) may not start with a zero "0" followed by another digit, and
- decimal numbers (with a decimal point) may not start or end with the decimal point or end with a zero following the decimal point.

For example, *version numbers* "09", "1.", ".5", and "1.670" are prohibited. The proper forms with the same numeric values are "9", "1", "0.5", and "1.67". The *user* may assign a *version* of "0" but not of "00".

Versions that are assigned automatically have additional restrictions:

- They shall be *numeric versions* excluding the decimal point and with a minimum value of "1".

- *Versions* shall be assigned incrementally. When assigning a *version* for a recipe with a given *class* and *name*, if no other recipe exists within the *namespace* with that *class* and *name*, then a *version* of "1" is assigned. Otherwise, the highest *existing version* already in use for that *class* and *name* is determined, and the new *version* is assigned a value equal to that value plus 1. For example, if the highest *version* in use has a numeric value of 5, then the next *version* assigned would be "6".

To compare *two versions*, they are converted to upper-case and compared character by character.

3.2.3.4 Recipe Identifiers — The **recipe identifier** is formed from the concatenation of the recipe's *class*, *name*, and *version*, in that order:

"/CLASS₁/CLASS₂/.../CLASS_n/NAME;VERSION"

where CLASS₁ is a *primary class* and CLASS_{i+1} is a *subclass* of CLASS_i. The recipe *name* follows *class*. *Name* and *version* are always separated by a semicolon ";" (3B₁₆).

Where CLASS_n is a unique class name, the form becomes

/CLASS/NAME;VERSION".

The recipe *identifier* is used for the OSS required attribute ObjID and shall conform to restrictions imposed on ObjID (see SEMI E39). The total length of ObjID may have additional restrictions imposed by the protocol.

NOTE: There is no necessary relationship between a recipe's *identifier* or *name* and any file name(s) under which the *body* and attributes may be stored internally. The recipe *identifier* is a logical reference to the recipe that is independent of specific platforms and implementations. In particular, file services provided by operating systems may have naming restrictions, such as length, that are incompatible with the requirements of RMS. A recipe may be stored internally in different ways, such as in a single flat file, a set of related files, or a database. Actual storage methods shall be invisible to RMS.

Where suppliers supporting recipes in *source form* also rewrite them, as discussed in Section 3.2.2.1.2, a method based on the recipe *identifier* (ObjID) is required to both distinguish between the two *forms* and to recognize the relationship between the original *source form* recipe and the derived *object form* recipe. The method may use any of the three elements of the



identifier for this purpose. For example, recipes in *object form* may be placed in a separate *subclass* called "/OBJ/", or a suffix such as ".obj" might be appended to the user-defined *name* of the original recipe.

3.2.3.4.1 Default Recipe Identifiers — It is always possible to reference a recipe by specifying its *full identifier* (*class*, *name*, and *version number*). In certain cases, described in Section 3.2.4.1.1, it may also be necessary to specify the *namespace* of a recipe. However, it is not always necessary to specify all components of the *identifier*. Specifically, *class* and/or *version* may be omitted when a recipe is *selected* for execution and within an *external reference*. *Namespace* is not normally specified. Rules define default values for *namespace* and *class*, and there are rules for determining the appropriate *version*.

The recipe *name* must always be specified.

When an *execution recipe* is *selected* for execution, the recipe's *class* defaults to "/PROCESS/".

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* (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 are *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 reusability 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 rec-

ipe 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 rule⁷ 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

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

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.

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 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.