



## **Recipe and Parameter (RaP) Management Usage Scenarios**

**International SEMATECH Manufacturing Initiative**  
Technology Transfer #08054929A-TR

**Advanced Materials Research Center, AMRC, International SEMATECH Manufacturing Initiative, and ISMI** are servicemarks of SEMATECH, Inc. **SEMATECH** and the **SEMATECH** logo are registered servicemarks of SEMATECH, Inc. All other servicemarks and trademarks are the property of their respective owners.

**Recipe and Parameter (RaP) Management Usage Scenarios**  
**Technology Transfer #08054929A-TR**  
**International SEMATECH Manufacturing Initiative**  
**May 6, 2008**

**Abstract:** This guideline provides basic usage scenarios and common behavior for systems utilizing Recipe and Parameter (RaP) Management capabilities as defined in SEMI E139. This models the anticipated mode of operation of ISMI member companies. The focus of the usage scenarios is normal operations. Exceptions are considered and discussed, but coverage of in-depth error handling use cases awaits more extensive feedback from the field. Definition of common usage is intended to result in more uniform implementations, leading to higher quality and lower cost of ownership.

**Keywords:** e-Manufacturing, Factory Automation, Process Control, Process Recipes, Recipes, Recipe Management

**Authors:** Lance Rist

**Approvals:** Steve Fulton, Project Manager  
Olaf Rothe, Program Manager  
Scott Kramer, Director, ISMI  
Sue Gnat, Technology Transfer Team Leader



## Table of Contents

1	EXECUTIVE SUMMARY .....	1
2	PURPOSE.....	1
3	SCOPE.....	2
4	REFERENCED DOCUMENTS .....	2
	4.1 SEMI Standards .....	2
	4.2 ISMI Documents .....	2
5	ACRONYMS, TERMINOLOGY, AND METRICS DEFINITIONS .....	2
	5.1 Acronyms .....	2
	5.2 Terminology .....	3
6	RAP USAGE SCENARIO OVERVIEW .....	3
	6.1 RaPnodes Defined in SEMI E139.....	3
	6.1.1 RaP Services Supporting RaP Functionality .....	4
	6.1.2 RaPnode Functionality .....	4
7	RAP PROTOCOL BINDINGS .....	5
	7.1 RaP Communication Services Mapped to SECS-II.....	5
	7.1.1 RaP Services Mapping to SECS-II .....	6
	7.1.2 TransferContainer Handling .....	6
	7.2 RaP Communication Services Mapped to XML/SOAP .....	6
8	RAP IMPLEMENTATION STRATEGIES .....	7
	8.1 Early RaP Implementation .....	7
	8.2 Next Generation .....	8
9	RAP USAGE SCENARIOS .....	10
	9.1 Overview .....	10
	9.2 Common Notes .....	10
	9.2.1 Conventions .....	10
	9.3 Fundamental Sequences.....	11
	9.3.1 getPDEdirectory() .....	11
	9.3.2 deletePDE() .....	13
	9.3.3 getPDEheader() .....	14
	9.3.4 getPDE() .....	15
	9.3.5 requestToSendPDE().....	17
	9.3.6 sendPDE() .....	18
	9.3.7 resolvePDE() .....	20
	9.3.8 verifyPDE() .....	22
	9.4 Unit Scenarios .....	24
	9.4.1 <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)> .....	24
	9.4.2 <E139-RMS-2 – Transfer Recipe to RMS (from Equipment)> .....	25
	9.4.3 <E139-RMS-3 – Non-Uploadable PDE Handling>.....	27
	9.4.4 <E139-RMS-4 – Equipment Recipe Store Change> .....	28
	9.4.5 <E139-RMS-5 – Recipe Download>.....	29
	9.4.6 <E139-RMS-6 – PDE Superseded>.....	30
	9.4.7 <E139-RMS-7 – Delete Recipe> .....	31
	9.4.8 <E139-RMS-8 – Edit Recipe>.....	32
	9.4.9 <E139-RMS-9 – Synchronize RMS and Equipment>.....	34

9.5	Processing Scenarios.....	35
9.5.1	<E139-JOB-1 – Process Job Setup> .....	35
9.5.2	<E139-JOB-2 – Secure Recipe Use>.....	37
9.6	Troubleshooting and Maintenance Scenarios .....	39
9.6.1	<E139-MNT-1 – Inspect Recipe>.....	39
9.6.2	<E139-MNT-2 – Validate Recipe>.....	40
9.6.3	<E139-MNT-3 – Software Update> .....	41
9.6.4	<E139-MNT-4 – Copy Recipe from Removable Media directly onto Tool> .....	42
10	OVERVIEW OVER SCENARIOS .....	43

### List of Figures

Figure 1	Interfaces Between RaPnodes .....	3
Figure 2	SECS II Communication Interfaces Between RaPnodes .....	5
Figure 3	SECS communication Path with Early Implementation Approach.....	7
Figure 4	XML Communication Paths with Next Generation Approach.....	9
Figure 5	getPDEdirectory().....	11
Figure 6	deletePDE() .....	13
Figure 7	getPDEheader() .....	14
Figure 8	getPDE() .....	15
Figure 9	requestToSendPDE() .....	17
Figure 10	sendPDE().....	18
Figure 11	resolvePDE() .....	20
Figure 12	verifyPDE() .....	22
Figure 13	E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS) .....	24
Figure 14	E139-RMS-2 – Transfer Recipe to RMS (from Equipment) .....	26
Figure 15	E139-RMS-3 – Large Recipe Handling .....	27
Figure 16	E139-RMS-4 – Equipment Recipe Store Change .....	28
Figure 17	E139-RMS-5 – Recipe Download.....	29
Figure 18	E139-RMS-6 – PDE Superseded .....	30
Figure 19	E139-RMS-7 – Delete Recipe.....	32
Figure 20	E139-RMS-8 – Edit Recipe.....	33
Figure 21	E139-RMS-9 – Synchronize RMS and Equipment.....	34
Figure 22	E139-JOB-1 – Process Job Setup.....	36
Figure 23	E139-JOB-2 – Secure Recipe Use .....	38
Figure 24	E139-MNT-1 – Inspect Recipe.....	39
Figure 25	E139-MNT-2 – Validate Recipe .....	40
Figure 26	E139-MNT-4 Copy Recipe from Removable Media Directly Onto Tool.....	42

## List of Tables

Table 1	RaPnode Service/Event Support .....	5
Table 2	Scenario Summary .....	44

## **Acknowledgments**

ISMI would like to acknowledge the work of Thomas Marsh, Konrad Rosenbaum, and Eberhard Teichmann of PEER Group in the creation of this document. Much of the work of creating this document is theirs. Also contributing ideas and review feedback were Steve Fulton, Toysha Walker, and Michael Conboy of ISMI.

## 1 EXECUTIVE SUMMARY

SEMI Standard E139, *Specification for Recipe and Parameter Management (RaP)*, specifies the interaction between the Factory Information and Control System (FICS) and the equipment to manage specifications (recipes) of equipment processing. RaP supports the following goals:

- Trusted recipe content, a guarantee that the recipe on the equipment is exactly the one the factory approved, downloaded, and/or selected
- Enhanced Process Control through flexible user defined parameter settings, and clear visibility of parameters and their defaults in recipe headers
- Reduced number of recipes due to enhanced parameterization opportunities and multi-part recipe component reuse
- Traceability of recipe and parameter usage through event reporting supported by guaranteed unique recipe identifiers
- Detailed and efficient verification process to ensure recipe validity
- Support of advanced FICS side recipe management approaches
- Flexible recipe editing allowing supplier specific recipe editors to be integrated into the factory infrastructure, paving the way for universal recipe editors in the future

With the goal of encouraging uniform adoption and implementation of RaP, this document defines common usage scenarios. Conformance to these usage scenarios reduces unnecessary variation in implementations. This benefits the implementers by reducing development effort and complexity, resulting in lower cost and higher quality. Support for testing is also made easier.

It should be noted that the processing scenarios are defined in the context of 300 mm standards and operations. However, they also apply equally well to 200 mm factories and are intended to apply to next generation factories.

## 2 PURPOSE

The Recipe and Parameter (RaP) usage scenarios defines a common approach to the use and implementation of RaP interfaces. These usage scenarios represent the anticipated behavior and mode of operation of the member companies of the International SEMATECH Manufacturing Initiative (ISMI). The usage scenarios are intended to guide device makers, equipment manufacturers, and software solution providers to a common usage approach, with the intent of simplifying the job of the implementer and tester. Accordingly, adherence to the usage scenarios ultimately will improve reliability, and reduce the cost of such implementations.

This document is not intended to define all possible scenarios that will be used in all situations. Individual device makers may have business needs that require additional, unique scenarios. The intent is to minimize the need for these additional scenarios and maximize the uniformity of implementations.

### 3 SCOPE

This document describes basic usage scenarios and common extended sequences of RaP behavior as specified in SEMI E139. Each RaP method description includes a summary of exception scenarios which may be generated during its use. Higher level, extended scenarios are also described on the basis of these fundamental sequences,.

Some extended scenarios include messaging and capabilities beyond the management of recipes. These additions are included to show the context of RaP scenarios within related activities. For example, recipe management is shown in the context of GEM 300 processing management.

These should not be taken to imply a dependence of RaP on other standards or a definition of those related scenarios.

### 4 REFERENCED DOCUMENTS

The following documents include applicable SEMI standards and other documents.

#### 4.1 SEMI Standards

- SEMI E5 – *SEMI Equipment Communication Standard 2 (SECS II)*
- SEMI E30 – *Generic Model for Communications and Control of Manufacturing Equipment (GEM)*
- SEMI E120 – *Common Equipment Model (CEM)*
- SEMI E139-1107 – *Recipe and Parameter Management (RaP)*
- SEMI E139.1-1106E – *XML Schema for the RaP PDE*
- SEMI E139.2-1107 – *SECS-II Protocol for Recipe and Parameter Management*

Note that SEMI E139.3, the mapping of RaP services to XML/SOAP, was in the SEMI standards approval process at the time this document was published and therefore was not yet available to the public.

#### 4.2 ISMI Documents

- [\*Recipe and Parameter \(RaP\) Management Supplier Guidance\*](#), ISMI, Technology Transfer #06104801A-ENG.

### 5 ACRONYMS, TERMINOLOGY, AND METRICS DEFINITIONS

#### 5.1 Acronyms

- ACL – Access Control List
- PDE – Process Definition Element
- FICS – Factory Information and Control System (ITRS term)
- RaP – Recipe and Parameter Management
- RMS – Recipe Management System
- uid – Unique Identifier

## 5.2 Terminology

The following terms are defined as they are used within this document.

PDE	– The smallest unit of a recipe that can be handled independently.
Recipe	– The collection of all PDEs that are needed to execute a process at the equipment. This can be a single executable PDE or a collection of PDEs.
TransferContainer	– Compressed collection of PDEs for transfer to/from a RaP-enabled application (Equipment, FICS, Recipe Editor)
RaP client	– An entity that is able to use the services provided by RaP. A single entity may be both a RaPnode and a RaP client.
RaPnode	– An entity/program that provides RaP services to RaP clients.

## 6 RAP USAGE SCENARIO OVERVIEW

This document provides basic usage scenarios and common sequences of a Recipe and Parameter Management interface based on SEMI E139. The scenarios provide guidance on the behavior of the entities participating in RaP communication.

### 6.1 RaPnodes Defined in SEMI E139

RaP specifies three participants in the management of recipes. These participants are each an instance of a SEMI E139 RaPnode and have a corresponding class name (Figure 1).

- **FICS** – representing the factory control systems, including RMS and equipment integration, and referred to as *FICSnode*
- **Equipment** – representing the manufacturing equipment, referred to as *EquipmentNode*
- **Recipe Editor** – representing an off-tool recipe editor, referred to as an *EditorNode*

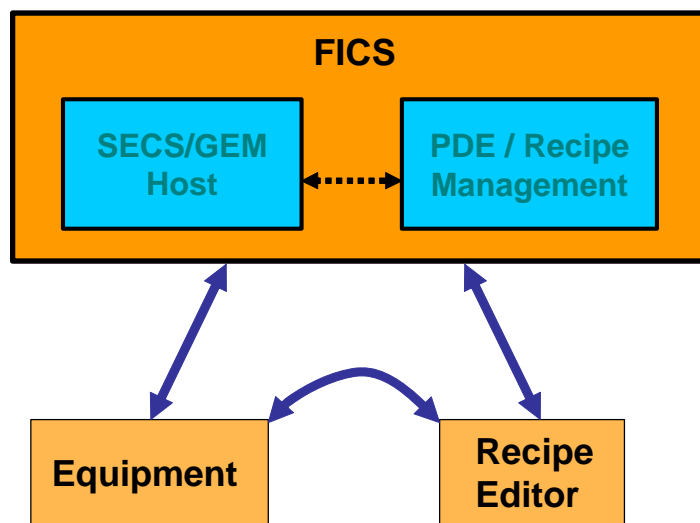


Figure 1 Interfaces Between RaPnodes

The RaP environment fulfills the following high level functionality:

- Execution of recipes
- Creation and editing of recipes
- Validation of recipes
- Tracking, transfer, and storage of recipes
- Event Notification

### 6.1.1 RaP Services Supporting RaP Functionality

The methods in the following list are specified by the RaP standard. Each method is listed with a high level summary of its function.

- *getPDEdirectory()* – This service requests a list of the PDEs managed by the RaPnode. The requestor may supply selection criteria (filters) for which PDEs to report and may also select certain attributes to be returned for each reported PDE.
- *getPDEheader()* – This service requests the service provider to return the complete header of the specified PDE(s).
- *getPDE()* – This service requests the service provider to return the specified PDE(s) in a TransferContainer.
- *requestToSendPDE()* – This service requests the service provider to grant or deny permission to transfer a collection of PDEs in a TransferContainer to the service provider.
- *sendPDE()* – This service requests the service provider to accept the set of PDEs in the included TransferContainer.
- *deletePDE()* – This service requests the service provider to delete the specified PDE(s) from the RaPnode.
- *resolvePDE()* – This service requests the service provider to determine the complete recipe hierarchy beneath the specified PDE based on the ReferencedPDEs.
- *verifyPDE()* – This service requests the service provider to check various aspects of the specified PDE.

The EquipmentNode must also report the following events via SECS-II Stream 6 event reporting and via E134 event reporting, if Interface A is supported:

- **PDEadded event** – Sent by the Equipment any time a PDE is added to its recipe store.
- **PDEremoved event** – Sent by the EquipmentNode any time a PDE is removed from its recipe store.

### 6.1.2 RaPnode Functionality

Each type of RaPnode provides a subset of the defined RaP services. Table 1 lists the RaP message services and events supported by each type of RaP node.

**Table 1 RaPnode Service/Event Support**

Service/Event Name	EquipmentNode	FICSnode	EditorNode
getPDEdirectory()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
getPDEheader()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
getPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
requestToSendPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
sendPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
deletePDE()	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
verifyPDE()	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
resolvePDE()	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
PDEadded event	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
PDEremoved event	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

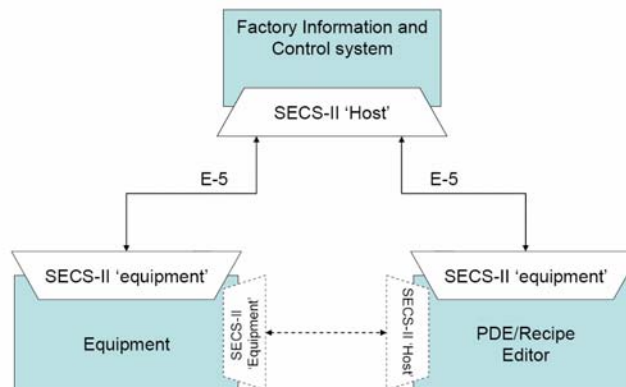
## 7 RAP PROTOCOL BINDINGS

RaP provides two protocol bindings: SECS-II and XML/SOAP. The first RaP interfaces are expected to use the SECS-II binding. Later equipment offerings of RaP are expected to supplement the SECS-II protocol with XML/SOAP messaging to support off-tool recipe editors and more sophisticated factory recipe management approaches. See Section 8 for discussion implementation strategies.

In some cases, the protocol binding transform a single service into a sequence of messages. This is not reflected in the scenarios defined in this document. The scenarios show only the SEMI E139 messages services used.

### 7.1 RaP Communication Services Mapped to SECS-II

The SECS-II binding for RaP is defined in SEMI E139.2. The message definitions are found in the SEMI E5 SECS-II standard as Stream 19. Figure 2 shows the communication links.



**Figure 2 SECS II Communication Interfaces Between RaPnodes**

While multiple SECS connections to the equipment are allowed by SEMI communications standards, this is not recommended. Off-tool editors are expected initially to utilize pre-existing, proprietary protocol mappings and move rapidly to use the XML/SOAP mapping described in Section 7.2.

### 7.1.1 RaP Services Mapping to SECS-II

The following list shows the SECS-II messages that correspond to each RaP service definition.

- getPDEdirectory() – S19,F1/F2
- deletePDE() – S19,F3/F4
- getPDEheader() – S19,F5/F6
- getPDE() – S19,F7/F8
- requestToSendPDE() – S19,F9/F10
- sendPDE() – S19,F11/F12 and S19,F13/F14
- resolvePDE() – S19,F15/F16
- verifyPDE() – S19,F17/F18

### 7.1.2 TransferContainer Handling

RaP defines the transfer of PDEs and PDEheaders in Transfer Containers using SEMI Stream 13 ‘Data Set Transfers’.

The service provider of the sendPDE() service initiates SEMI Stream 13 DataSetTransfer messaging to transfer the TransferContainer from the client.

The service provider of getPDE() and getPDEheader() services initiate SEMI Stream 13 DataSetTransfer messaging to transfer the TransferContainer to the client.

## 7.2 RaP Communication Services Mapped to XML/SOAP

**Note:** At the time of publication, SEMI E139.3 has not received final approval from SEMI. It currently exists as SEMI draft document #3981. Approval in 2008 is anticipated.

SEMI E139.3 will provide XML/SOAP over HTTP as an additional protocol binding for RaP services. SEMI E139.3 communication is established on a SEMI E132 port (specification for Equipment Client Authentication and Authorization) which can be shared with other SEMI applications, such as SEMI E134 Data Collection.

The XML/SOAP mapping supports multiple clients. This is ideal for off-tool recipe editors, which may be running on the desktop computers of multiple engineers for a given equipment.

XML/SOAP also provides a communication link separate from SECS-II. This allows an option for the factory to make an independent connection to the equipment for recipe transfer, reducing the interference with the command and control messaging flowing over the SECS-II link.

## 8 RAP IMPLEMENTATION STRATEGIES

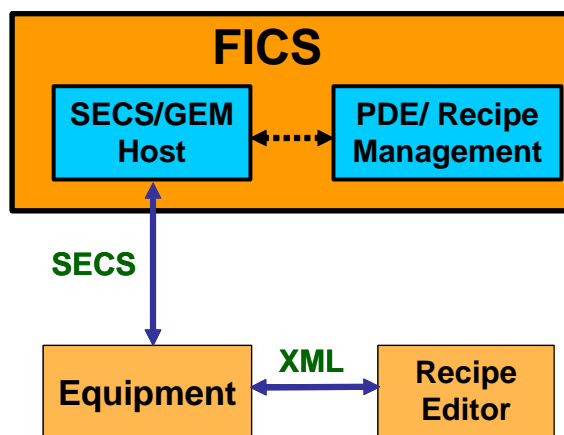
To provide context for the scenarios defined in this document, this section describes two implementation strategies for the RaP capabilities defined by SEMI E139. The first is a model likely to be supported in early RaP installations, and the second represents a future or next generation model of implementation. Both strategies rely on RaP capabilities for all RaPnodes, but differ in the internal implementation of the nodes. The difference between the two strategies is rooted in the role of the factory system (FICS) components.

It is important to note that the usage scenarios described in this document depict the FICSnode as a black box. In this section, to help explain these two implementation strategies, the varying roles of the SECS/GEM Host and the recipe management system (RMS) are discussed.

Primitive approaches to recipe management that neither track nor store equipment recipes at the factory level are not considered in this document. However, some scenarios found in this document will apply to that case.

### 8.1 Early RaP Implementation

It is anticipated that the first implementations of RaP will represent a modest deployment of RaP capabilities. This “early implementation” model proposed here is analogous to common recipe management practices currently in place. Recipe-related communications from factory to equipment remain the exclusive responsibility of the SECS/GEM Host and are exchanged using the SECS protocol (see Figure 3).



**Figure 3 SECS communication Path with Early Implementation Approach**

Control of the equipment recipe store is maintained by the factory, but continues to be very tactical. Recipes are downloaded only when immediately needed for the current processing task (see Section 9.5.1). Recipes are deleted from the equipment only when a user decides to do so and often only in reaction to low disk space warnings from the equipment.

Recipe download is significantly reduced, however, based on the security provided by remote checksum verification and by the reduction in overall recipes made possible by expanded recipe parameter availability and improved recipe component reuse. Recipe download will become rare with reductions from previous download frequency potentially in excess of 95%. This will result in a corresponding reduction in processing delay on the equipment caused by recipe download.

This approach assumes that the “golden copy” of each recipe/PDE is stored in the factory RMS. In the few cases where upload of a PDE is impractical due to its size or because of equipment restrictions, the PDE is assumed to be tracked within the RMS, utilizing the PDEheader information.

The factory RMS is a passive participant in this approach. It reacts only to service requests and does not initiate service requests to other factory components. The RMS does provide the centralized storage of recipe bodies as well as maintaining meta-data about those recipes such as approval status. The SECS/GEM Host is the primary client of the RMS, storing and retrieving recipes.

This implementation strategy assumes that the use of recipe parameters to modify recipes at run-time will become much more extensive. However, the uses will follow the typical patterns seen prior to RaP—a single source of parameters for a given equipment, primarily from run-to-run controller implementations.

Off-tool recipe editors will be common and should constitute the first widespread use of the XML/SOAP protocol for RaP. These off-tool recipe editors are assumed to be proprietary and communicate directly to the equipment to store newly created recipes/PDEs and to fetch recipes/PDEs for editing. These editors will not typically communicate directly to the factory system (for example, to the RMS).

In the rare case where the XML/SOAP protocol is used for factory to equipment communications, it will most likely be implemented from SECS/GEM Host as a direct replacement of the SECS equivalent messaging (for example, to relieve SECS communication overload). The particular protocol used in this case will not be reflected in the scenarios, which show only the messaging, not the protocol used.

## 8.2 Next Generation

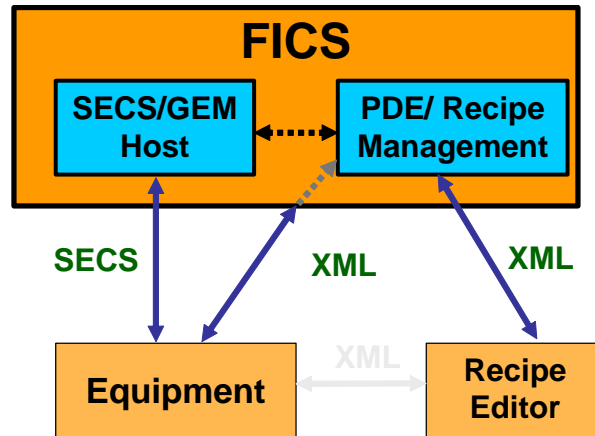
Future “Next Generation” implementations will implement a more advanced strategy for deployment of the RaP standards. This model features an elimination of the just-in-time recipe download and gives the RMS a more active and intelligent role in recipe management.

In this approach, the primary responsibility for managing the equipment recipe store moves from the SECS/GEM Host to the RMS. The SECS/GEM Host will continue to use RaP services in the context of process setup. It will ensure that the needed recipe is on the tool and verified to be correct and complete. However, it will not upload or download recipes. This communication will likely continue to use the SECS protocol.

Recipe upload and download between factory and equipment will be initiated by the RMS. The RMS will become an XML/SOAP client of the equipment, using this alternate communication link to remove all SECS link congestion due to recipe transfer (see Figure 4).

Once again, the golden copy of each recipe is stored in the RMS (except where upload is not practical). The RMS will actively manage the equipment recipe store using a longer time horizon. The goal will be to ensure that all recipes needed for processing will be on the tool before they are needed. This will eliminate the remaining process setup delay due to recipe transfer. The RMS will also directly delete unneeded recipes when they are superseded or retired.

The factory RMS will periodically initiate download of needed recipes to the equipment based on factory business rules. For example, recipe download could be driven by a scheduler, which could predict the most likely processes to occur within the next day/week/month.



**Figure 4 XML Communication Paths with Next Generation Approach**

Periodically, the RMS will re-synchronize the equipment recipe store with the list of recipes expected to be in the store. In this process, any PDEs unregistered with the RMS are detected and dispositioned.

Use of recipe parameters will continue to expand to support new control approaches and multiple sources for parameter settings. This predicts the need for a factory component responsible for merging/arbitrating settings from these different sources into a single set of values for a given process job.

The use of multi-part recipes will increase due to increasing equipment complexity and the representation of more process settings as PDEs (what are today tuning constants, equipment constants, images, etc.).

The combination of multi-part recipes and recipe parameters means that the record of which recipe components and settings were actually used during a job is difficult to determine. The use of traceability events that report each time a recipe component is executed (wafer by wafer) and the parameter values used in each case solves this problem. Traceability events will come into common use with this implementation strategy.

Off-tool editors will continue to be commonly used. The first, crude “universal” editors will begin to appear. Universal editors are those that can edit recipes from multiple suppliers’ equipment.

In this implementation strategy, off-tool editors will typically communicate with the RMS to store newly created recipes/PDEs and to fetch recipes/PDEs for editing. Communication directly to the equipment has inherent risks to equipment performance due to transport of large recipes and compute intensive validation processes. It also places unknown and unapproved recipes on the equipment. It is expected that download to the equipment will become exclusive to the RMS, establishing a more clear “single point of control” for the recipe store. When the RMS receives a new recipe from the editor, it may apply application specific logic before downloading the recipe to the equipment, such as engaging an approval process for the recipe or determining the optimal time to deploy the recipe.

It is recognized that RaP implementations may contain elements of both the “early implementation” strategy and the “next generation” strategy, according to the needs of the user and as a migration strategy between the two.

The scenarios defined below will identify which implementation strategy it supports (or both) and, where both, any small differences in application for each. Table 2 summarizes the scenarios and shows which scenarios support each implementation strategy.

## 9 RAP USAGE SCENARIOS

### 9.1 Overview

The RaP usage scenarios defined in this document can be classified in the following groups:

- **Fundamental Sequences**
  - Small reusable functional cases
  - Represented by the defined SEMI E139 services
- **Unit Scenarios**
  - Represented by a combination of fundamental sequences from the SEMI E139 standard

### 9.2 Common Notes

All scenarios share the pre-condition that a communication link is established and ready to be used.

Communication failures can occur at any time. Usually these abort the scenarios and adequate steps should be taken to mark the executed action as aborted and to inform the user about the problem.

Most RaP services can return negative results (for example, if resolvePDE fails to resolve a GID, verifyPDE returns a failure code). It is typical to abort a scenario when an error is reported. There are exceptions to that approach (for example, temporary conditions, branching situations, and partial failures). Exception handling approaches are discussed with each scenario.

During the initial phase of commercialization it should be expected that error codes evolve as more is learned about recipe handling and exceptional situations. The list contained in this document is subject to change where differences occur; the SEMI E139 standard should be followed.

#### 9.2.1 Conventions

The usage scenarios are presented in a common format. Each usage scenario has the following structure:

- Introductory paragraph describing the service, including a UML sequence diagram detailing the low level messaging of the scenario
- **Service Parameters** – a listing of the parameters and values to be provided upon invocation of services (provided for Fundamental Sequences only)
- **Response** – a description of the possible responses upon service invocation (provided for Fundamental Sequences only)

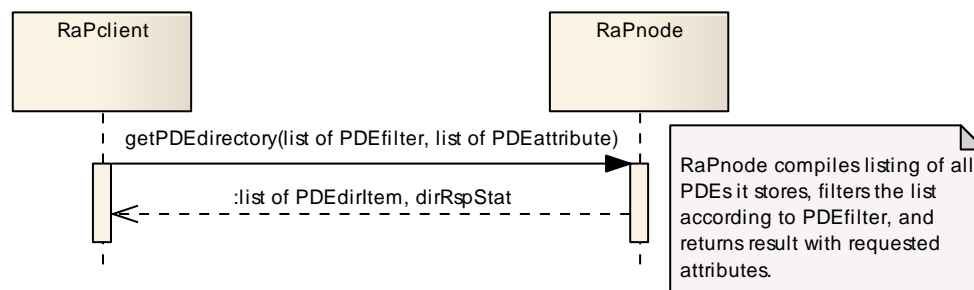
- **Pre-conditions** – a list of all pre-conditions that must be satisfied in order to initiate the scenario (provided only for Unit Scenarios)
- **Post-conditions** – a list of all post-conditions that must be satisfied to ensure that scenario has completed successfully (provided only for Unit Scenarios)
- **Notes** – key information of particular interest. For the Fundamental Sequences this includes a list of the Unit Scenarios where the sequence is used
- **Exceptions** – a listing of the possible error conditions and exceptions that can occur during the scenario execution

### 9.3 Fundamental Sequences

Clients of RaP services can be any entity with communication access to a RaPnode. The following fundamental use cases represent the services, defined in SEMI E139. They may be reused and combined into more complicated use cases.

#### 9.3.1 getPDEdirectory()

The getPDEdirectory() service is used to request a list of PDEs maintained by the service provider (see Figure 5). The requestor may supply a filter that will yield a subset of the full list of PDEs. The requestor may also specify which attributes it is interested in. This service must be provided by all RaPnodes (EquipmentNode, FICSnode, and EditorNode).



**Figure 5** getPDEdirectory()

##### 9.3.1.1 getPDEdirectory() Service Parameters

The getPDEdirectory() service has two parameters. The first is a list of PDEfilters, which is a set of selection criteria. The criteria consist of a PDEAttributeName, an operator (“EQ,” “GT,” “Like,” etc.), and a PDEAttributeValue. A complete list of the possible filter operators is available in SEMI E139. If no filters are supplied, all PDEs on the equipment will be returned.

The second parameter to getPDEdirectory() is a list of PDEAttributes. This list specifies the attributes from selected PDEs that will be returned in the response to the getPDEdirectory() invocation.

##### 9.3.1.2 getPDEdirectory() Response

In the response, each PDEDirItem contains a uid representing the PDE, and a list of PDEAttribute/PDEAttributeValue pairs.

The following attributes may be requested:

- uid
- name
- gid
- groupName
- description
- type
- executable
- createDate
- createNode
- author
- userInfo – supplied value matched to any userInfo string in the PDE
- supplierInfo – supplied value matched to any supplierInfo string in the PDE

The parameter dirRspStat returns the status response for the getPDEdirectory() service.

The following status codes are defined:

- OK – successfully returned request data.
- BadFilter – one or more of the PDE filters were not properly specified.
- BadAttribute – one or more of the requested attributes do not exist.
- Other – a problem occurred that is not described by the above values.

### 9.3.1.3 getPDEdirectory() Notes

getPDEdirectory() is used in the following scenarios:

- Section 9.4.1 – <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)>
- Section 9.4.3 – <E139-RMS-3 – Non-Uploadable PDE Handling>
- Section 9.4.8 – <E139-RMS-8 – Edit Recipe>
- Section 9.4.9 – <E139-RMS-9 – Synchronize RMS and Equipment>
- Section 9.6.1 – <E139-MNT-1 – Inspect Recipe>

While for some scenarios this service is used to find out which PDEs are stored on the RaPnode, it is often used as input for a user selection. In the latter case it is recommended that at least the “name”, “groupName,” and “description” attributes are requested and subsequently shown to the user. Local preferences are likely to add “author” and “createDate,” among others.

The filter attribute in the request message can be used to narrow the list of returned PDEs. Useful applications include:

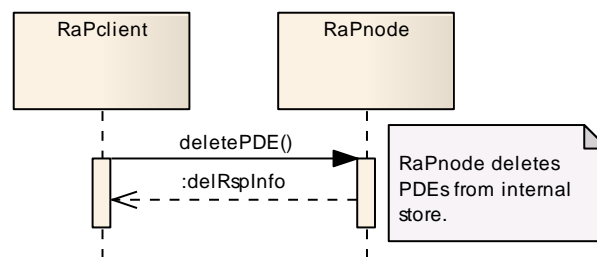
- Confirming the existence of a single PDE by filtering by its uid.
- Confirming the existence of a PDE group by filtering by its gid.
- An engineer searching for specific recipe using a “Like” filter on the name attribute.

#### 9.3.1.4 getPDEdirectory() Exceptions

Assuming that the service is implemented correctly at the receiving RaPnode, any response other than “OK” usually points to a problem with the request. These situations will need the attention of the engineer responsible for the client.

#### 9.3.2 deletePDE()

The deletePDE() service is used to request that specified PDEs be deleted from the service provider (see Figure 6). This service must be provided by the EquipmentNode and the EditorNode. The FICSnode is not required to provide this service.



**Figure 6** deletePDE()

##### 9.3.2.1 deletePDE() Service Parameters

The deletePDE() service takes a list of uid as a parameter. This list specifies the uids of the PDEs to be deleted and must contain at least one uid.

##### 9.3.2.2 deletePDE() Response

The parameter delRspInfo contains a list of (uid, delRspStat) for each PDE that was requested to be deleted. The parameter delRspStat returns the service status code that describes whether the deletion was successful.

The following status codes are defined:

- OK – the PDE was successfully deleted.
- PDEnotFound – the requested PDE did not exist on the RaPnode.
- PDElocked – the PDE cannot be deleted at the moment. It may be reserved for processing.
- Other – a problem occurred that is not described by the above values.

### 9.3.2.3 deletePDE() Notes

The deletePDE() service is used in the following scenarios:

- Section 9.4.6 – <E139-RMS-6 – PDE Superseded>
- Section 9.4.7 – <E139-RMS-7 – Delete Recipe>
- Section 9.5.2 – <E139-JOB-2 – Secure Recipe Use>

The deletePDE() service handles PDEs independently. It may succeed for some PDEs and fail for others.

### 9.3.2.4 deletePDE() Exceptions

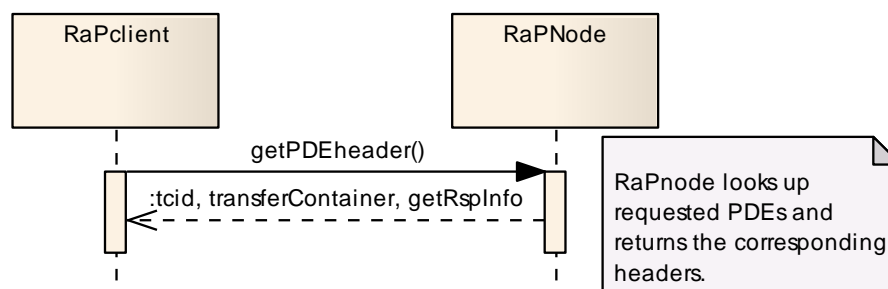
Note that delRspInfo is given individually for each PDE to be deleted. Some may succeed and others fail. Any code other than “OK” means the PDE was not deleted.

The status code “PDEnotFound” signals that the PDE was no longer stored on the receiving RaPnode when the call was received. The desired result was achieved. The fact that a PDE the client believed to be on the RaPnode did not exist may be interpreted as an error condition or be ignored.

The “PDElocked” status code may indicate a temporary situation. The request may be rescheduled for a later time. Failure to delete would not normally prevent subsequent action. The status code “Other” should be interpreted as a fatal failure of the service. A human should inspect the associated description to evaluate the problem.

### 9.3.3 getPDEheader()

PDEheaders are transferred in TransferContainers, addressed by a tcid (see Figure 7). Prior to sending a response, the service provider is responsible for locating the PDEs requested, extracting the headers, packaging them in a TransferContainer, and generating a tcid for the TransferContainer.



**Figure 7** `getPDEheader()`

This service must be provided by all RaPnodes (EquipmentNode, FICSnode, and EditorNode).

Please note that while `getPDEheader()` is consistently shown as only one message throughout this document the SECS-II binding (SEMI E139.2) implements it with a more complex series of several messages. See SEMIE139-1107, Section 6.5.5.

### 9.3.3.1 getPDEheader() Service Parameters

The getPDEheader() service takes a list of uid as a parameter. This list specifies the uids of the PDEs for which the PDE headers shall be returned in the TransferContainer.

### 9.3.3.2 getPDEheader() Response

The parameter getRspInfo contains a list of (uid, getRspStat). The parameter getRspStat returns the service status code. Note that it is possible to receive only a subset of the requested PDEs. The transferContainer contains the PDE headers that were successfully retrieved.

The following status codes are defined:

- OK – headers have been returned with the message.
- PDEnotFound – the requested PDE did not exist at the RaPnode.
- PDElocked – the requested PDE cannot be delivered at the moment.
- Other – a problem occurred that is not described by the above values.

### 9.3.3.3 getPDEheader() Notes

The getPDEheader() service is used in the following scenarios:

- Section 9.4.3 – <E139-RMS-3 – Non-Uploadable PDE Handling>

### 9.3.3.4 getPDEheader() Exceptions

Expected status codes:

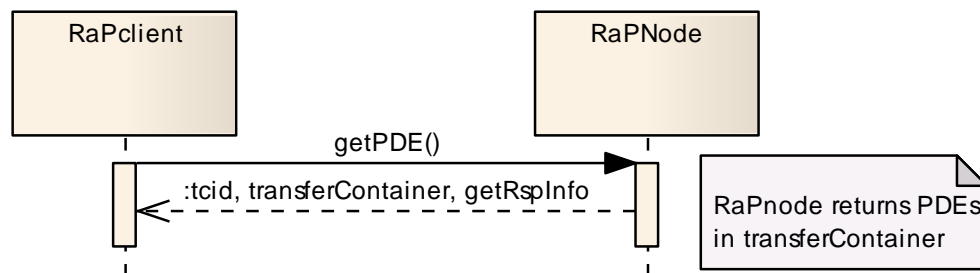
- The “PDElocked” code should never be received for getPDEheader(). However, if it should be received, the client has the option to try again later.

Status codes interpreted as fatal errors:

- All other status codes (except “OK”)

### 9.3.4 getPDE()

PDEs are transferred in TransferContainers, addressed by a tcid (see Figure 8).



**Figure 8**      **getPDE()**

Prior to sending a response, the service provider is responsible for locating the PDEs requested, packaging them in a TransferContainer, and generating a tcid for the TransferContainer.

This service must be provided by all RaPnodes (EquipmentNode, FICSnode, and EditorNode).

Please note that while getPDE() is consistently shown as only one message throughout this document the SECS-II binding (SEMI E139.2) implements it with a more complex series of several messages. See SEMI E139-1107, Section 6.5.5.

#### **9.3.4.1 getPDE() Service Parameters**

The getPDE() service takes a list of uid as a parameter. This list specifies the uids of the PDEs that shall be returned in the TransferContainer.

#### **9.3.4.2 getPDE() Response**

The parameter getRspInfo contains a list of (uid, getRspStat). The parameter getRspStat returns the service status code. Note that it is possible to receive only a subset of the requested PDEs. The TransferContainer contains the PDEs that were successfully retrieved.

The following status codes are defined:

- OK – headers have been returned with the message.
- PDEnotFound – the requested PDE did not exist at the RaPnode.
- PDElocked – the requested PDE cannot be delivered at the moment.
- Other – a problem occurred that is not described by the above values.

#### **9.3.4.3 getPDE() Notes**

The getPDE() service is used in the following scenarios:

- Section 9.4.1 – <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)>
- Section 9.4.8 – <E139-RMS-8 – Edit Recipe>
- Section 9.4.9 – <E139-RMS-9 – Synchronize RMS and Equipment>
- Section 9.6.1 – <E139-MNT-1 – Inspect Recipe>

#### **9.3.4.4 getPDE() Exceptions**

Please note that the status code is listed in the response once for each requested PDE, hence the call can succeed for some PDEs while failing for others.

Expected status codes:

- The “OK” status code means that the request succeeded for the referenced PDE.
- If the “PDElocked” code is received, the client has the option to try again later.

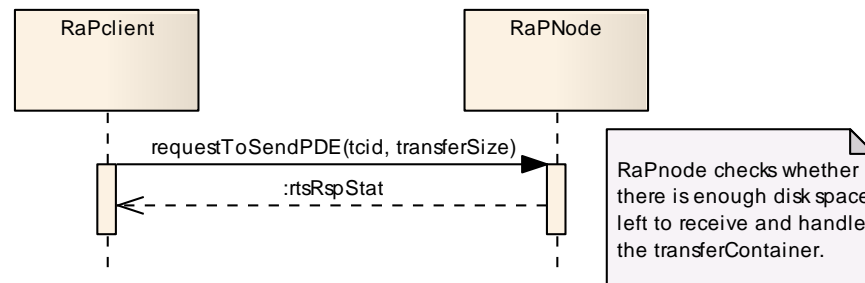
Status codes interpreted as fatal errors:

- The “PDEnotFound” code will be reported if the PDE is not available at the RaPnode. This code is fatal unless there is another entity that transfers PDEs to this node and has not finished yet.
- The “Other” status code will be reported for any other problem.

### 9.3.5 requestToSendPDE()

The requestToSendPDE() requests permission to send a TransferContainer containing PDEs to the service provider (see Figure 9). The size of the proposed TransferContainer is provided so that the service provider can ensure that room is available.

This service must be provided by all RaPnodes (EquipmentNode, FICSnode, and EditorNode).



**Figure 9** requestToSendPDE()

#### 9.3.5.1 requestToSendPDE() Service Parameters

The requestToSendPDE() service takes two parameters. The first is the tcid which identifies the TransferContainer to be transferred. The second parameter is the transferSize which specifies the TransferContainer size in bytes.

#### 9.3.5.2 requestToSendPDE() Response

The parameter rtsRspStat returns the service errors:

The following status codes are defined:

- OK – permission to send has been granted.
- NoResources – the RaPnode does not have enough resources to deal with the request (for example, it is out of disk space).
- Other – a problem occurred that is not described by one of the status codes above.

#### 9.3.5.3 requestToSendPDE() Notes

The requestToSendPDE() service is used immediately before sendPDE() in the same scenarios (see Section 9.3.6.3).

This service is optional, but it should be treated as required. The client should use requestToSendPDE() for a pre-check of whether the receiver can handle the subsequent sendPDE call. For example, a TransferContainer which is sufficiently large may cause memory overflows as well as issues with disk space.

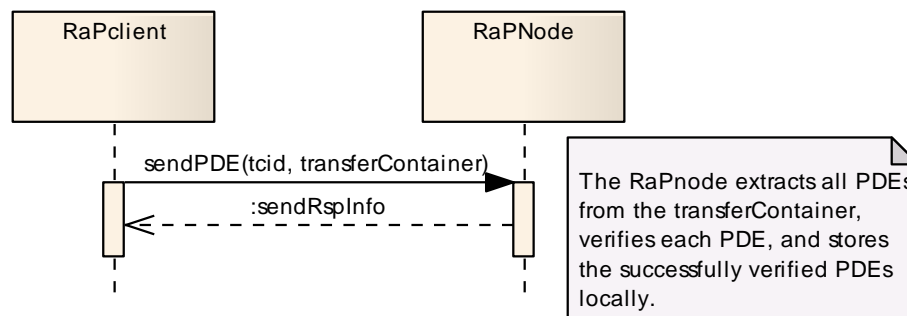
This service can, if it succeeds, guarantee that the TransferContainer can be stored on the receiving node. The subsequent sendPDE() might still fail for other reasons, such as verification failure.

### 9.3.5.4 requestToSendPDE() Exceptions

The “OK” status code means that the subsequent sendPDE() is permitted. The “No Resources” return code can indicate a temporary or indefinite problem, depending on what specific condition caused the error. All other status codes should be interpreted as fatal errors.

### 9.3.6 sendPDE()

The sendPDE() service is used to transfer PDEs contained in a TransferContainer to the service provider (see Figure 10). This service must be provided by all RaPnodes (EquipmentNode, FICSnode, and EditorNode).



**Figure 10** sendPDE()

If this service is requested on an EquipmentNode, the EquipmentNode must verify each PDE upon receipt. This verification must be the equivalent of that initiated by the verifyPDE() service with verifyType = validity and verifyDepth = single.

Please note that while sendPDE() is consistently shown as only one message throughout this document the SECS-II binding (SEMI E139.2) implements it with a more complex series of several messages. See SEMI E139-1107, Section 6.5.6.

#### 9.3.6.1 sendPDE() Service Parameters

The sendPDE() service takes two parameters. The first is the tcid which identifies the TransferContainer to be transferred. The second parameter is the transferContainer which contains the PDE to be sent.

#### 9.3.6.2 sendPDE() Response

The parameter sendRspInfo contains a list of (uid, sendRspStat, verifyRspStat).

The parameter sendRspStat returns the service status codes. The following status codes are defined:

- OK – PDE has been accepted.
- NoResources – the RaPnode ran out of resources to process the request (for example, disk space).
- TargetMismatch – the recipient is not compatible with this PDE’s ExecutionTarget.

- PDElocked – the PDE already exists at the receiver and cannot be overwritten at the moment (for example, it may be in use).
- VerificationFailed – the PDE failed to verify.
- Other – any problem not covered by the above codes.

The parameter verifyRspStat returns the verification status. The following status codes are defined:

- OK – verification was successful.
- NotFound – the PDE was not found.
- ChecksumFail – the computed checksum did not match the checksum in the PDE.
- SyntaxError – the syntax of the PDE header or body is not correct.
- ContentError – the content of the PDE did not match allowed values.
- InvalidInputMap – the input map supplied contains errors.
- ResolveDenied – the equipment is not allowed to resolve PDEs.
- NoExecutionTarget – no execution target in the equipment matches the one requested in the PDE.
- OutputParameterError – the parameters have not been applied correctly to referenced PDEs.
- ReferencedPDEnotFound – one or more referenced PDEs could not be found.
- NoVerification – the receiving RaPnode does not perform verification.
- Other – any problem not described by the above status codes.

### 9.3.6.3 sendPDE() Notes

The sendPDE service is used in the following scenarios:

- Section 9.4.2 – <E139-RMS-2 – Transfer Recipe to RMS (from Equipment)>
- Section 9.4.5 – <E139-RMS-5 – Recipe Download>
- Section 9.4.6 – <E139-RMS-6 – PDE Superseded>
- Section 9.4.8 – <E139-RMS-8 – Edit Recipe>
- Section 9.4.9 – <E139-RMS-9 – Synchronize RMS and Equipment>
- Section 9.5.1 – <E139-JOB-1 – Process Job Setup>
- Section 9.5.2 – <E139-JOB-2 – Secure Recipe Use>

If the receiver of the service is an EquipmentNode, it is expected that it performs a validation, equivalent to verifyPDE with verifyType=validity and verifyDepth=single, of each received PDE. It is recommended to send an additional verifyPDE with verifyDepth=all for all Master PDEs in order to check that propagated parameters are in the correct range.

### 9.3.6.4 sendPDE() Exceptions

sendPDE() has two exception fields, sendRspStat for overall status and verifyRspStat for the verification portion (a subordinate status value). The expected status codes are:

- EquipmentNode: sendRspStat returns “OK” and verifyRspStat returns “OK.”
- FICSnode or EditorNode: sendRspStat returns “OK” and verifyRspStat returns “OK” or “NoVerification.”

Note that for FICSnode and EditorNodes, a “NoVerification” response for verifyRspStat is normal and not considered an error condition.

Status codes interpreted as fatal errors:

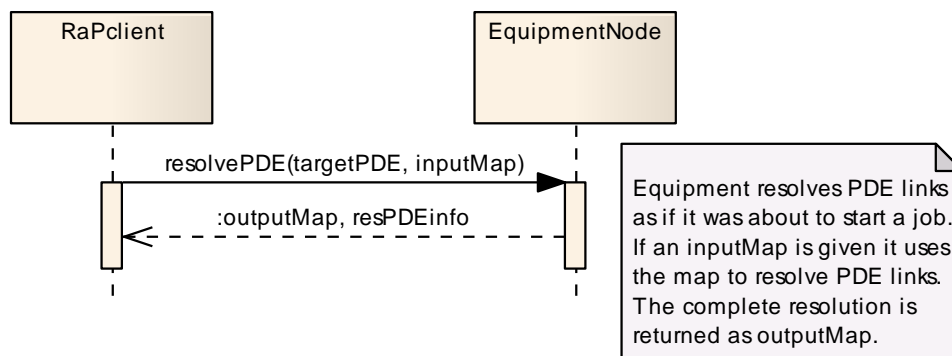
- All other combinations of sendRspStat and verifyRspStat

Status codes that should never appear in response to sendPDE:

- NotFound
- InvalidInputMap
- ResolveDenied
- ReferencedPDEnotFound

### 9.3.7 resolvePDE()

In the recipe hierarchy, each reference of a PDE is either a uid or a gid value. All gids must be resolved to uids before a recipe can be executed. The resolvePDE() service allows the client to check whether a recipe can be successfully resolved by the equipment (see Figure 11).



**Figure 11** resolvePDE()

The resolvePDE() service is supported only by the EquipmentNode. The EquipmentNode must resolve the recipe hierarchy exactly the same as if the recipe was to be executed at the time of the resolvePDE() request.

### 9.3.7.1 resolvePDE() Service Parameters

The service parameter targetPDE is the uid of the PDE to be resolved.

The service parameter inputMap is a list of pdeRef-resolution pairs, where the pdeRef contains the reference to a PDE, the resolution contains the value of the uid attribute of the PDE to which a PDE reference is resolved.

### 9.3.7.2 resolvePDE() Response

The parameter outputMap is a list of pdeRef, resolution pairs. Each pair is a reference to a PDE and the uid that is the resolution for that reference.

The parameter resPDEInfo contains a list of (pdeRef, resPDEstat). All references found in the recipe structure are represented by a pdeRef. Each such reference has one resPDEstat value assigned, which represents the service status code.

The following status codes are defined:

- OK – no problems encountered.
- MissingTargetPDE – the targetPDE itself could not be resolved by any PDE on the equipment.
- MissingMapPDE – a needed PDE specified by the inputMap was not found on the equipment.
- MissingReferencedPDE – a referenced PDE could not be found on the equipment.
- InvalidInputMap – the inputMap contained errors.
- ResolveDenied – the equipment is configured not to resolve PDE references and not all references were either uid references or listed in the inputMap.
- Other – any problem not covered by the status codes above.

### 9.3.7.3 resolvePDE() Notes

The resolvePDE() service is used in the following scenarios:

- Section 9.4.1 – <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)>
- Section 9.4.3 – <E139-RMS-3 – Non-Uploadable PDE Handling>
- Section 9.4.8 – <E139-RMS-8 – Edit Recipe>
- Section 9.5.1 – <E139-JOB-1 – Process Job Setup>

It is recommended that ResolvePDEreferences is set to FALSE, in order to force the FICS to resolve all references in a PDEmap/inputMap. This ensures that no unexpected PDEs are used by the equipment.

For a few special cases it may make sense to set ResolvePDEreferences to TRUE. If ResolvePDEreferences is set to TRUE, the EquipmentNode is responsible for the final resolution of any gid's not resolved by the client in the inputMap.

The outputMap from resolvePDE() can be used as an inputMap for a subsequent resolvePDE() or as the PDEmap supplied as a parameter to a Process Job. In many cases, such a PDEmap is stored in the factory recipe management system as part of the definition of a recipe.

### 9.3.7.4 resolvePDE() Exceptions

Expected status codes:

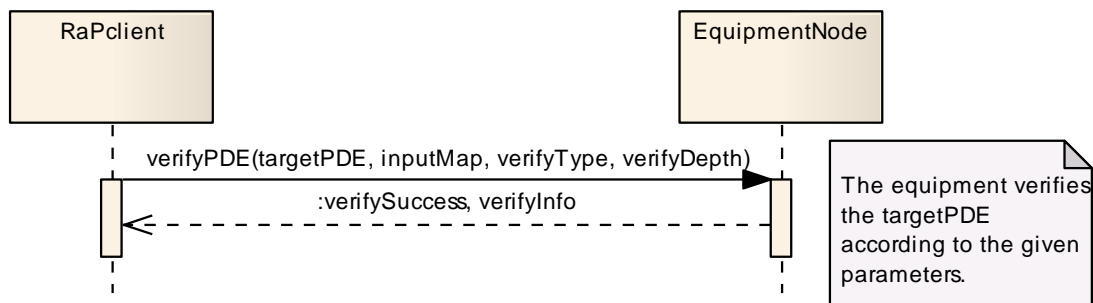
- “OK” represents the only non-exceptional state.
- “MissingTargetPDE,” “MissingMapPDE,” and “MissingReferencedPDE” may be resolved by downloading the missing PDE and retrying the transaction.

Status codes interpreted as fatal errors:

- All other status codes (except “OK”)

### 9.3.8 verifyPDE()

The verifyPDE() service is used to check the validity of a specified PDE. Multiple levels of verification are available as defined in the parameter verifyType (see Figure 12). The verifyPDE() service is supported only by the EquipmentNode.



**Figure 12** verifyPDE()

#### 9.3.8.1 verifyPDE() Service Parameters

The service parameter `targetPDE()` contains the uid of the PDE to be resolved.

The service parameter `inputMap` is a list of `pdeRef`, resolution pairs where the `pdeRef` contains the reference to a PDE, the resolution contains the value of the uid attribute of the PDE to which a PDE reference is resolved.

The service parameter `verifyType` specifies one of the following verification types:

- Checksum (verifies checksum of each PDE and all related external PDEbodies)
- Validity (syntax, checksum and all criteria, provided by the equipment)

The service parameter `verifyDepth` specifies one of the following verification scopes:

- Single (the specified PDE only)
- All (all referenced PDEs in the recipe structure are verified, including whether all required parameters are supplied)

#### 9.3.8.2 verifyPDE() Response

The parameter `verifySuccess` returns “true” if no error occurred during verification, otherwise it returns “false”.

The parameter `verifyInfo` is a list of (`uid`, `verifyRspStat`) pairs. Each `uid` has one `verifyRspStat` value assigned, which represents the verification status.

The following status codes are defined:

- `OK` – verification was successful.
- `NotFound` – the PDE was not found.
- `ChecksumFail` – the computed checksum did not match the checksum in the PDE.
- `SyntaxError` – the syntax of the PDE header or body is not correct.
- `ContentError` – the content of the PDE did not match allowed values.
- `InvalidInputMap` – the input map supplied contains errors.
- `ResolveDenied` – the equipment is not allowed to resolve PDEs.
- `NoExecutionTarget` – no execution target in the equipment matches the one requested in the PDE.
- `OutputParameterError` – the parameters have not been applied correctly to referenced PDEs.
- `ReferencedPDEnotFound` – one or more referenced PDEs could not be found.
- `NoVerification` – the receiving RaPnode does not perform verification.
- `Other` – any problem not described by the above status codes.

### 9.3.8.3 `verifyPDE()` Notes

The `verifyPDE()` service is used in the following scenarios:

- Section 9.4.5 – <E139-RMS-5 – Recipe Download>
- Section 9.5.1 – <E139-JOB-1 – Process Job Setup>
- Section 9.5.2 – <E139-JOB-2 – Secure Recipe Use>
- Section 9.6.1 – <E139-MNT-2 – Validate Recipe>

The `verifyPDE()` service contains all functionality of `resolvePDE` if called with `verifyDepth=all`.

This service can consume a significant amount of time and processing resources when called with `verifyType=validity`, depending on the complexity of the implementation and the tested PDEs. It is recommended to call `verifyPDE()` with `verifyType=validity` only once per PDE and equipment instance, preferably when the equipment is idle.

### 9.3.8.4 `verifyPDE()` Exceptions

Expected status codes:

- “OK” represents the only non-exceptional state
- “NotFound” and “ReferencedPDEnotFound” may be resolved by downloading the missing PDE and retrying the transaction.

Status codes that should never appear in response to `verifyPDE`:

- No Verification

Status codes interpreted as fatal errors:

- The “ChecksumFail” status code should not appear for any PDE that has been successfully downloaded to the tool. Hence it most probably signals serious problems with the recipe storage memory (failing hard disk or RAM) or some accidental or willful alteration of a PDE.
- The “SyntaxError” status code can signal various problems. For example, it might indicate a PDE that is incorrectly structured or intended for a different equipment type, or it might indicate an equipment whose software has changed and is now incompatible with the previously existing PDE format.
- The “ContentError” and “OutputParameterError” status codes can occur for PDEs that have been altered outside the equipment if the editor did not recognize the correct limits for parameters or that limits have changed on the equipment (due to hardware or software changes).
- The “ResolveDenied” status code can signal that the equipment is incorrectly configured (if automatic resolution is desired) or that resolution pairs are missing from the inputMap.
- The “NoExecutionTarget” status code can signal that the equipment hardware has been altered or that the PDE has been sent to the wrong equipment.

## 9.4 Unit Scenarios

### 9.4.1 <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)>

The introduction of a recipe into the RMS may also be triggered by the RMS or any other component of the FICS (see Figure 13). Depending on organizational needs this scenario may be preferred for some IC-makers.

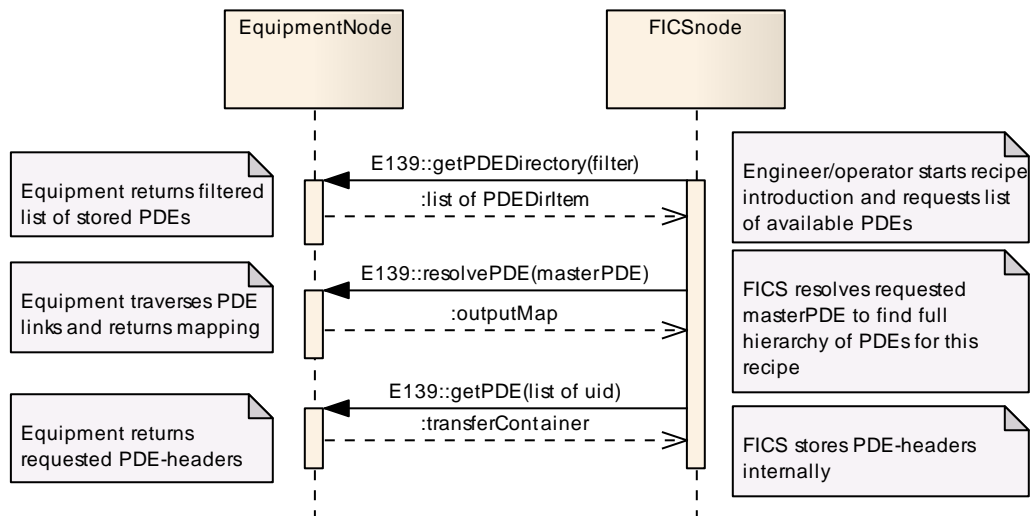


Figure 13 E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)

The FICS will first need to query the equipment for existing PDEs and then give the user a choice based on pre-selected filter criteria. Afterwards the FICS will request the Master PDE of the recipe from the equipment. If the equipment uses a hierarchy of PDEs to represent recipes, the FICS will have to find out what exact PDEs are referenced by the Master PDE.

#### **9.4.1.1 Pre-Conditions**

A new PDE/recipe is available on the equipment.

#### **9.4.1.2 Post-Conditions**

The new recipe has been transferred to the FICS.

#### **9.4.1.3 Notes**

The selection of filters for getPDEdirectory depends on the specific needs of the FICS user. Typical usage may be the specification of PDEs created after a specific date or ones whose name contains specific sub-strings.

If the FICS keeps track of all changes to the tools recipe store it may be possible to offer a list of new PDEs to the user without querying first. However, it is likely that at least some additional information may be necessary to allow the user to choose the correct PDE. It is possible to retrieve most information with getPDEdirectory.

If the equipment uses a single PDE to represent each recipe, then it will not be necessary to use resolvePDE to get the uids of all sub-PDEs.

If the equipment uses large PDEs, it may be advisable to query the PDEs in smaller groups or even one-by-one, hence using multiple getPDE transactions.

#### **9.4.1.4 Exceptions**

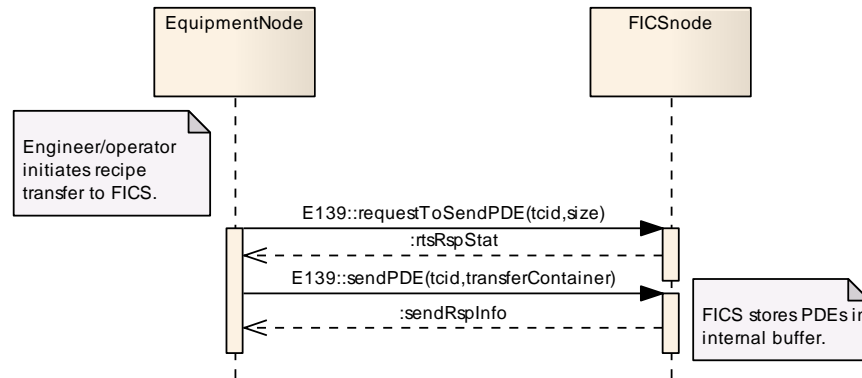
Any of the used calls can fail due to communication problems or missing permissions. In that case the scenario should be aborted.

### **9.4.2 <E139-RMS-2 – Transfer Recipe to RMS (from Equipment)>**

When a new recipe has been created or an existing recipe altered<sup>1</sup> and is ready to be introduced to the RMS, the operator or engineer will usually send it from the equipment over an existing connection with the FICS to the RMS storage area (see Figure 14). Later it will be classified and approved inside the FICS for its proper use.

---

<sup>1</sup> Any alteration of a PDE will result in a new PDE.



**Figure 14 E139-RMS-2 – Transfer Recipe to RMS (from Equipment)**

This scenario is considered to apply to early implementations of RaP, but not to the Next Generation approach. Recipe creation is expected to transition to off-tool recipe editors that will transfer the PDEs to the equipment via the FICS. See Section 9.4.8 (<E139-RMS-8 – Edit Recipe> scenario) for more on editor initiated recipe transfers).

#### 9.4.2.1 Pre-Conditions

A new PDE (or recipe) on the equipment is ready to be introduced into the RMS.

#### 9.4.2.2 Post-Conditions

A copy of the PDE (or all PDEs of the recipe) is stored in the input buffer of the RMS waiting to be accepted into permanent storage.

#### 9.4.2.3 Notes

This scenario is recommended over the scenario in Section 9.4.1 (<E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)>), since it is more convenient for the operator or engineer working on the recipe.

Since the RMS will typically require more information in order to import a new PDE/recipe into the system, it is recommended that the received PDE/recipe be placed in a temporary storage buffer. The user can then retrieve it from this buffer during the RMS import process.

This is a user-initiated backup of an equipment recipe to the FICS. It can be used to temporarily backup a recipe that is in development, even if it is not ready to be added into the permanent factory store of equipment recipes.

#### 9.4.2.4 Exceptions

The requestToSendPDE() and sendPDE() calls can fail either because of connection problems or because necessary permissions are lacking. In both cases the scenario should simply be aborted and the user notified via the user interface. No automatic recovery is necessary or possible.

### 9.4.3 <E139-RMS-3 – Non-Uploadable PDE Handling>

Under some circumstances transferring the complete PDE is not practical, but central tracking of the PDE is still desired. Since RaP guarantees the integrity of a PDE on an equipment, it is safe to use only the header information for tracking PDEs.

This variation of scenario <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)> (Section 9.4.1) uses `getPDEheader` instead of `getPDE` to retrieve the PDE information without a `PDEbody` (see Figure 15). Otherwise the two scenarios are identical.

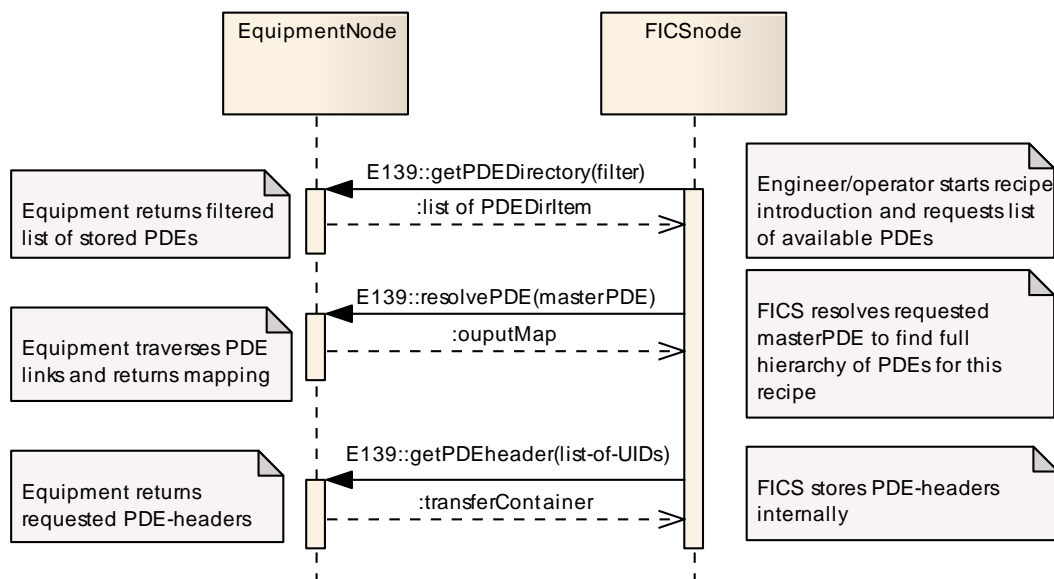


Figure 15 E139-RMS-3 – Large Recipe Handling

#### 9.4.3.1 Pre-Conditions

New non-uploadable PDE on the equipment is ready to be introduced into the FICS.

#### 9.4.3.2 Post-Conditions

The FICS will know the header information (meta-data) of the new PDE and be able to track it. However, the actual recipe body will still only be resident on the equipment.

#### 9.4.3.3 Notes

Use of this scenario involves the risk that a PDE/recipe is known to and expected by the FICS, but deleted on the equipment. In that case the FICS will not be able to restore the PDE/recipe on the equipment. Alternate backup strategies should be considered.

This scenario may be preferable to the full upload of recipes in cases when the full upload would take too long and/or would require too much storage space.

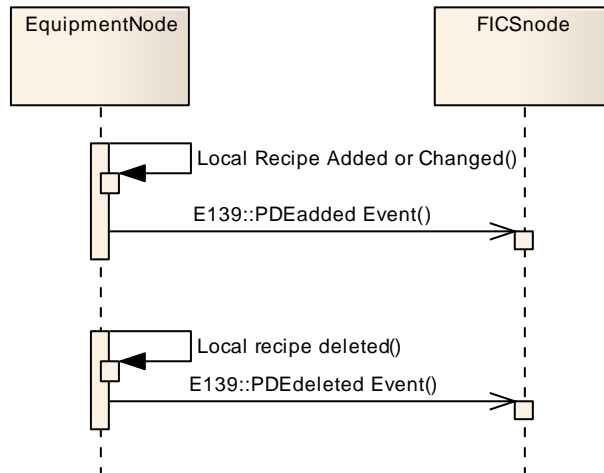
#### 9.4.3.4 Exceptions

Any of the used calls can fail due to communication problems or missing permissions. In that case the scenario should be aborted.

#### 9.4.4 <E139-RMS-4 – Equipment Recipe Store Change>

For advanced Recipe Management Systems (RMS) it is important to be informed about any change in the recipe storage area of production equipment (see Figure 16).

If a new PDE is created or an existing PDE is altered and subsequently saved as a new PDE the equipment will send a “PDEadded” event. If a PDE is deleted it will send a “PDEremoved” event. This gives the FICS a chance to react to the new situation, for example, by resending the lost PDE.



**Figure 16 E139-RMS-4 – Equipment Recipe Store Change**

This scenario is considered to apply to Next Generation application of RaP. Early implementations are not expected to track the equipment recipe store closely. Instead, they will focus on the presence of certain required recipes.

##### 9.4.4.1 Pre-Conditions

None.

##### 9.4.4.2 Post-Conditions

The PDE was added or removed and the FICS was informed about this action.

##### 9.4.4.3 Notes

This scenario is not enough to track completely all recipe changes, since it relies on a functional communication between equipment and FICS at the time the change is made. It is recommended to request a full PDE directory at communication startup, upon host startup or after communication interruptions, by using the `getPDEdirectory()` call without filters. See Section 9.4.9 (<E139-RMS-9 – Synchronize RMS and Equipment>) for a corresponding scenario.

These events occur both for local deletion/addition of PDEs as well as remotely initiated actions. Note that there may be multiple RaP clients via the XML/SOAP interface. For example, this scenario this would inform the FICS when a recipe editor deletes a PDE from the equipment.

These events can be received via E30 events (S6F11) and/or Interface-A. In both cases the FICS has to subscribe to these events and to connect appropriate data to it.

It is worth noting that a given factory may react differently to this notification. For example, the FICS may automatically upload any new/unknown PDE that appears on the equipment (see Section 9.4.1 – <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)>).

- If the factory automatically uploads unknown recipes/PDEs, this supports a temporary backup of new recipes on the equipment, including those that are in development and not ready for introduction to the permanent factory store of equipment recipes—see Section 9.4.9 (<E139-RMS-9 – Synchronize RMS and Equipment>).

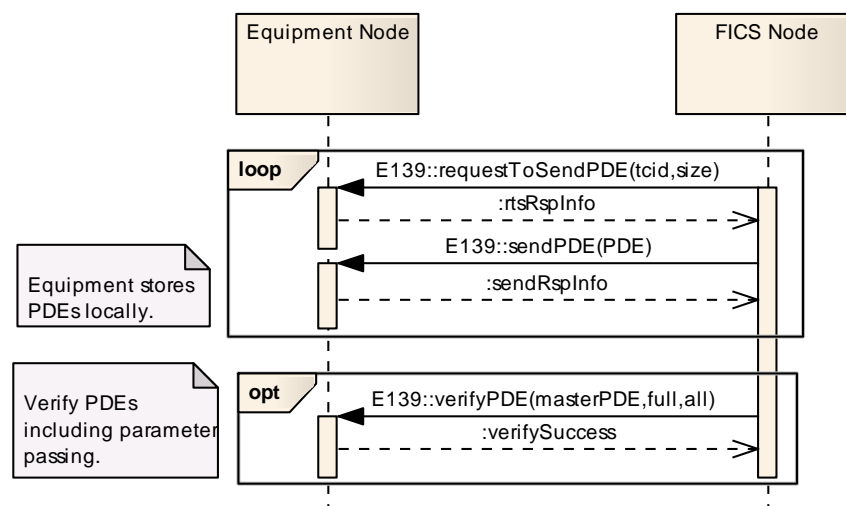
#### 9.4.5 <E139-RMS-5 – Recipe Download>

When the FICS decides that a specific recipe is needed on the equipment, it initiates the download of that recipe.

Since RaP provides a level of confidence in the integrity of recipes previously downloaded to the equipment, recipe download can be separated into two categories. The first, represented by this scenario, covers download of a complete recipe to an equipment where no version of that recipe currently exists. The second is represented by the scenario <E139-RMS-6 – PDE Superseded> and covers the case of a new version of a recipe where one or more of the PDEs constituting that recipe are replaced with new ones.

It is expected that there will be a migration from just-in-time recipe download to a Next Generation look-ahead approach that eliminates processing delays waiting for recipe transfer. In either case, the scenario used is the same.

In this scenario, all PDEs of the recipe are transmitted first, then optionally the recipe is verified (see Figure 17). The equipment is required to verify each PDE after it receives it automatically. The additional full verification shown below determines whether the sent recipe as a whole is intact and compatible with the equipment.



**Figure 17 E139-RMS-5 – Recipe Download**

### 9.4.5.1 Pre-Conditions

The FICS determines that a recipe previously not stored at the equipment is now needed for future process.

### 9.4.5.2 Post-Conditions

The new recipe is stored at the equipment.

### 9.4.5.3 Notes

If the sum of all PDEs to be uploaded for the intended Master PDE is very large, it might be necessary to split the transfer into several smaller blocks and use sendPDE() on each of those.

It is recommended to verify the recipe with verifyDepth=all and verifyType=checksum after it is downloaded. This step can be omitted on equipment that uses only a single PDE per recipe, since each PDE by itself must be verified when received by the equipment.

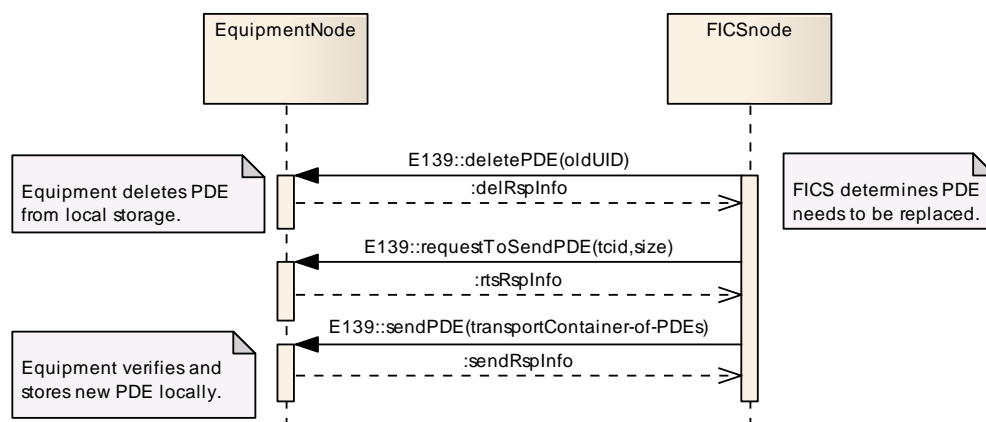
If one of the PDEs is already stored at the tool this will not have an influence on the success of the scenario, since equipment are required to accept the new PDE version silently.

### 9.4.5.4 Exceptions

Any call in this scenario can fail. In this case the scenario should be aborted.

## 9.4.6 <E139-RMS-6 – PDE Superseded>

When a recipe or recipes are updated, it may be embodied in changes to any or all of the PDEs that make up that recipe (see Figure 18). One updated PDE may affect multiple recipes. Once a recipe change has been made, the FICS must ensure that the new version of the recipe will be used in place of the old one. This will typically mean downloading the new PDEs and removing<sup>2</sup> the old ones from each affected equipment.



**Figure 18 E139-RMS-6 – PDE Superseded**

<sup>2</sup> In some factories, recipes are managed in sets, so that particular material experiences the recipes from the same set. In this case, new recipes and old recipes may be used simultaneously and the download/deletion may be separated.

This is considered a Next Generation scenario by virtue of its intent to remove old recipes that it deems unnecessary. Early implementations of RaP are more likely to use the <E139-RMS-5 – Recipe Download> scenario as needed and download any changed PDEs during processing setup (see <E139-JOB-1 – Process Job Setup> scenario).

#### **9.4.6.1 Pre-Conditions**

A new or updated recipe has been approved for use on an equipment or group of equipment.

#### **9.4.6.2 Post-Conditions**

The PDEs representing the old version of the recipe have been removed from the equipment.

The PDEs representing the new version of the recipe are resident on the equipment.

#### **9.4.6.3 Notes**

Depending on whether the equipment represents recipes as one single PDE or a hierarchy of PDEs, deletePDE() will be used with a single uid or a list of uids. Likewise, the TransferContainer of the sendPDE() will contain one PDE or multiple PDEs.

A new version of a PDE never overwrites an old version, since their uids are different.

This scenario is repeated for each applicable equipment.

#### **9.4.6.4 Exceptions**

If deletePDE fails in any way a warning should be sent to a user to investigate the problem, but the scenario should not be stopped. See Section 9.3.2.4 for details on how to handle these error codes.

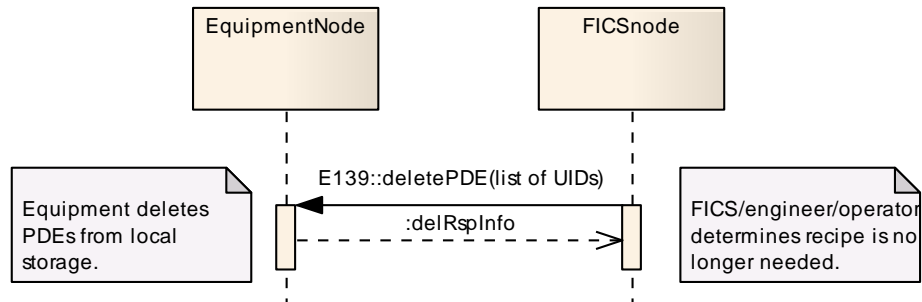
If requestToSendPDE() or sendPDE() fails, the scenario should be aborted and appropriate action be taken to ensure that the equipment is brought back to a usable state. Usually this will require analysis of the problem by a human being.

#### **9.4.7 <E139-RMS-7 – Delete Recipe>**

At some point, a recipe (or PDE) will need to be removed for an equipment (see Figure 19). The removal may be needed to free space for other recipes. Alternately, the recipe may have been retired or superseded. In this case, it should be removed to prevent accidental use.

Several PDEs can be deleted with only one message. These can be PDEs belonging to the same recipe that is scheduled to be removed. Other situations for this scenario include “monthly house cleaning”, removal as any individual PDE is superseded, or no longer needed, or pruning to make room on the equipment (for example, PDE is still valid, but not needed for a while).

This scenario is expected to be used primarily with early implementations of RaP. It assumes that users execute the deletion process manually. In Next Generation implementations of RaP, the RMS will track which recipes are needed on the equipment and which are not. User input will be made to the RMS. Deletions will occur automatically (for example, during the <E139-RMS-9 – Synchronize RMS and Equipment> scenario).



**Figure 19 E139-RMS-7 – Delete Recipe**

#### 9.4.7.1 Pre-Conditions

The engineer/operator/FICS determines that a PDE (or list of PDEs) is no longer needed at the equipment.

#### 9.4.7.2 Post-Conditions

The unwanted PDEs are no longer present on the equipment.

#### 9.4.7.3 Notes

Usually all PDEs belonging to the same recipe will be deleted at the same time with only one call to deletePDE().

The FICS should make sure that the deletion request does not conflict with other requirements (for example, no PDEs that are still needed for production should be deleted).

The FICS should make sure that the user requesting the deletion has sufficient privileges inside the FICS to execute this action, since it can potentially be disruptive to the process.

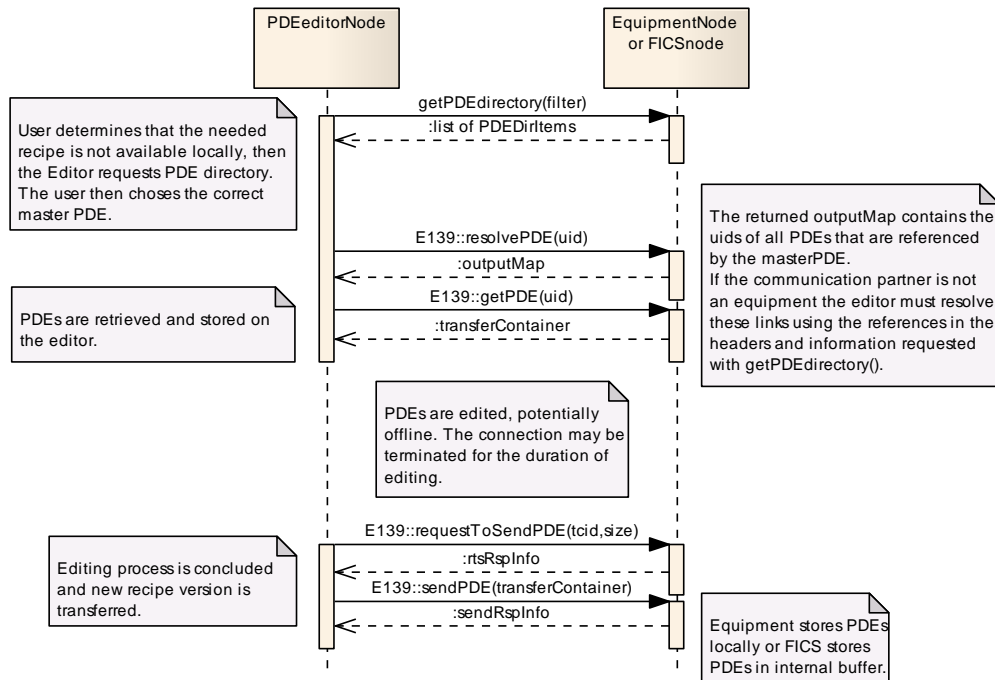
#### 9.4.7.4 Exceptions

The deletePDE() call can fail. One example is when a PDE is currently locked by a running process. In this case a warning message should be sent to a user to diagnose the problem.

However, if the return code is PDEnotFound, the “error” can be safely ignored since the PDE has already been removed prior to the call of deletePDE().

### 9.4.8 <E139-RMS-8 – Edit Recipe>

Remote recipe editors can be utilized to edit recipes without blocking the equipment during that time (see Figure 20). In early implementations of RaP, the editor is expected to communicate with the equipment for recipe/PDE exchange. Next Generation implementations will migrate to interact with the FICS for recipe/PDE exchange.



**Figure 20 E139-RMS-8 – Edit Recipe**

First the editor needs to retrieve the PDE and its linked PDEs. After editing all changes will be stored in new PDEs on the editor and will be sent back to the originating node using `sendPDE`.

This scenario may be used to edit a single PDE or all PDEs of a recipe.

#### 9.4.8.1 Pre-Conditions

The PDEs to be edited must be available on a RaPnode accessible by the editor.

The user must be able to identify the PDEs to be edited (for example, the Master PDE of a recipe that can then be resolved).

#### 9.4.8.2 Post-Conditions

The new PDEs have been transferred to the Equipment/FICS node.

- If the FICS is the target node, a copy of the PDE (or all PDEs of the recipe) is stored in the input buffer of the RMS waiting to be accepted into permanent storage.
- If the equipment is the target node, the PDEs are stored directly in permanent storage.

#### 9.4.8.3 Notes

In most cases it will be possible to transfer all changed PDEs with just one call to `sendPDE()`. However, it is possible to transfer multiple PDEs of the same recipe in smaller batches or one by one, if large PDE size makes this necessary.

Significant time can lapse during the editing of the recipe. If a connection oriented binding (for example, SECS-II) of SEMI E139 is used, it might be advisable to break and later reestablish the connection. Both sides should be able to handle a communication break between parts of the scenario.

### 9.4.8.4 Exceptions

If any of the used calls fails the scenario should be aborted.

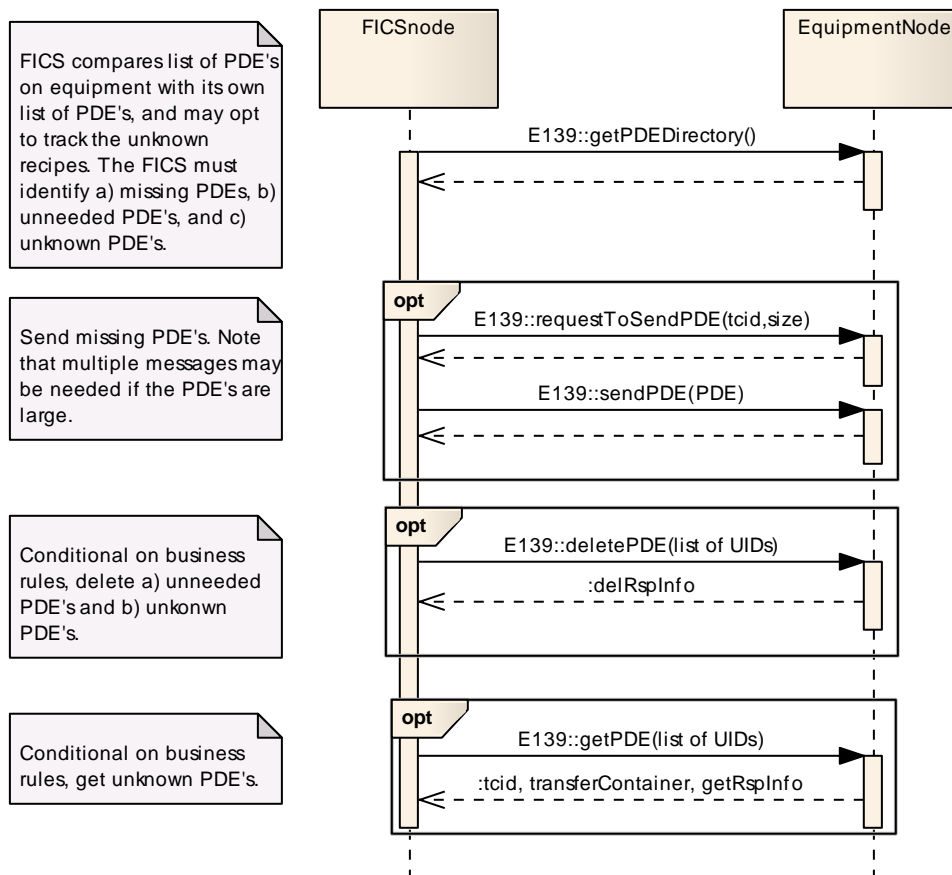
### 9.4.9 <E139-RMS-9 – Synchronize RMS and Equipment>

This scenario can be part of the connection start up between FICS and equipment or it can be triggered by other means (see Figure 21). It makes sure that all needed recipes are available at the equipment for later processing.

The first stage is to query the equipment for all PDEs that are currently stored. In stage two all missing PDEs are sent to the equipment. Optionally, the PDEs can be verified to make sure parameters are correctly passed down from the Master PDE to sub-PDEs.

The selection of PDEs to send depends on the FICS. Most commonly, all PDEs that are approved for process at this equipment instance and are needed for upcoming jobs will be chosen.

This scenario is expected to be commonly used with Next Generation implementations of RaP. Early RaP implementations are not expected to track the equipment recipe store closely.



**Figure 21 E139-RMS-9 – Synchronize RMS and Equipment**

#### 9.4.9.1 Pre-Conditions

Some reason exists to doubt that the factory's list of recipes/PDEs on this equipment is correct (equipment startup, communication interruption, etc.).

#### 9.4.9.2 Post-Conditions

The equipment has all appropriate PDEs to meet current factory needs.

#### 9.4.9.3 Notes

It is the FICS' responsibility to select the PDEs that are sent to the tool and to partition them into the TransferContainer. Therefore, it is possible that all PDEs are transported in the same container, they are grouped into manageable amounts, or even transported one by one. Depending on this selection, the loop might be executed once or several times.

Note that the SEMI E139 standard does not require all PDEs belonging to one recipe to be transferred in the same transport container or in a particular order.

The FICS may encounter unknown recipes in the list returned by `getPDEdirectory()`. It may choose to delete them, ignore them, log their presence, or upload them into the FICS. In the former case, the scenario <E139-RMS-7 – Delete Recipe> would be used. In the latter case, the scenario <E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)> is appropriate.

- If the factory automatically uploads unknown recipes/PDEs, this supports a temporary backup of new recipes on the equipment, including those that are in development and not ready for introduction to the permanent factory store of equipment recipes (see <E139-RMS-4 – Equipment Recipe Store Change> scenario).

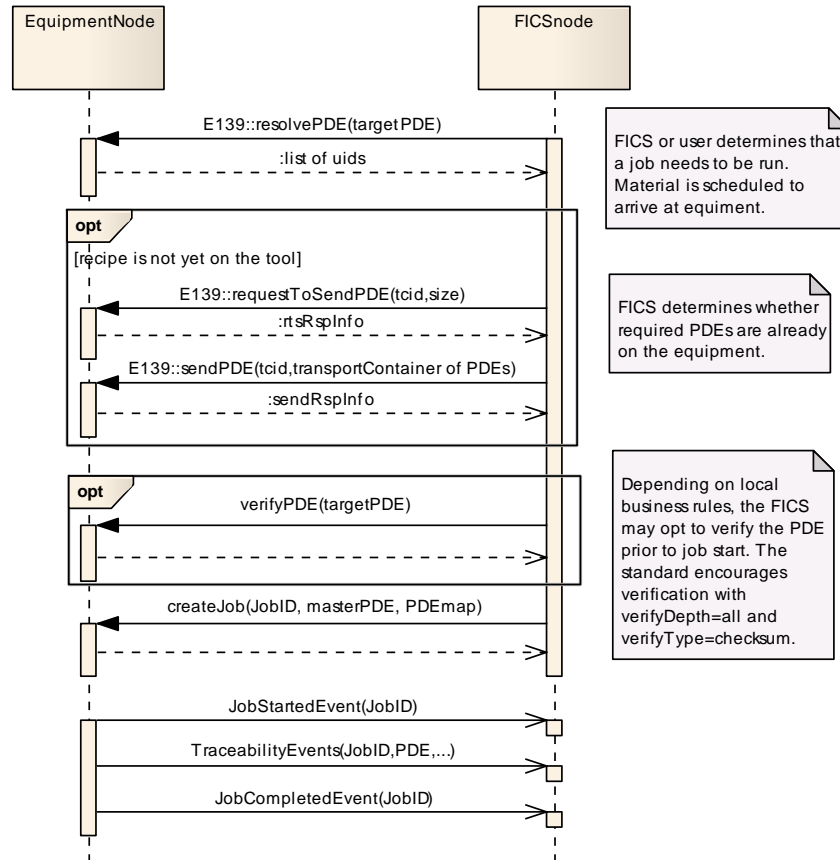
#### 9.4.9.4 Exceptions

Any call in this scenario can fail. In this case the scenario should be aborted. Whether further steps are taken depends on the needs of the surrounding FICS environment. It is possible to continue with the equipment if the already existing PDEs are enough for some processes, while it is also thinkable that more drastic measures are taken.

### 9.5 Processing Scenarios

#### 9.5.1 <E139-JOB-1 – Process Job Setup>

This scenario represents a typical process scenario utilizing RaP (see Figure 22). When the FICS determines that a job needs to be started on the tool, it first needs to find out whether the necessary recipe (list of PDEs) is available at the equipment. This may be accomplished either by



**Figure 22 E139-JOB-1 – Process Job Setup**

querying real-time information from its own list of recipes or by requesting this information from the tool directly. If the recipe is not yet on the tool it must be transferred before the job can be set up and started.

#### 9.5.1.1 Pre-Conditions

The tool must be in a state ready to set up and run jobs. The necessary recipe may or may not be on the tool at the time the scenario starts.

#### 9.5.1.2 Post-Conditions

The job is completed. The recipe remains on the tool awaiting the next job requiring it.

#### 9.5.1.3 Notes

The first message (getPDEdirectory) is intended to find out whether the required PDE is on the tool and may be optional in environments that keep track of every change in the recipe directory.

Overwriting existing PDEs in order to prevent/reverse inadvertent change is no longer necessary with RaP, since SEMI E139 mandates that the uid changes with each change of the PDE.

In early implementations the recipe download will occur via the SECS-II protocol from the SECS/GEM host system. In the Next Generation approach it is anticipated that this

communication will move to the SOAP/XML protocol with a direct connection between the RMS and the equipment, while the job management remains on the SECS/GEM host.

The equipment is encouraged to verify the full PDE hierarchy (verifyDepth=All, verifyType=Checksum) used for a job before the recipe is used each time unless it is confirmed that the equipment does this automatically.

The createJob call, JobStartedEvent and JobCompletedEvent are here for illustration, in practice they need to be replaced by whatever mechanism is used to set up and run jobs. Typically this will facilitate SEMI E40 and E94 Jobs. This scenario is intended to augment the 300 mm scenarios, not to create conflicts.

The TraceabilityEvents encompass all events that inform the FICS about the processing state of the Job/PDE. These include events that tell the FICS about the start and end of process steps, as well as events informing the FICS about changes in PDE parameters.

It is also possible to request the checksum value in the getPDEdirectory() and compare that to the checksum of the version stored in the FICS, preventing accidental or intentional replacement of the PDE.

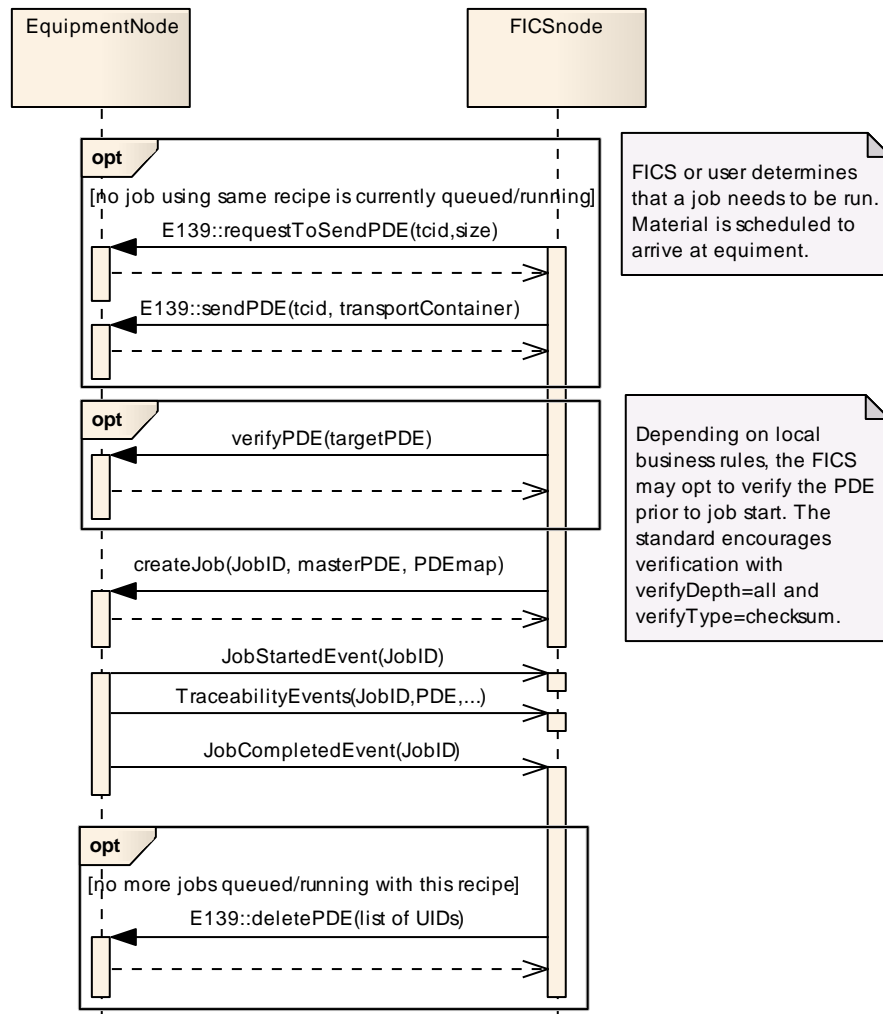
#### **9.5.1.4 Exceptions**

If the getPDEdirectory() request fails to execute (that is, it returns errors), it is safe to assume the PDE is not on the tool and overwrite it.

If the requestToSendPDE() or sendPDE() request fails, the scenario must be aborted.

#### **9.5.2 <E139-JOB-2 – Secure Recipe Use>**

In some environments, such as a joint venture running multiple proprietary processes, it may be desirable to extend a special protection to the intellectual property represented by a recipe by limiting the amount of time each recipe exists in the storage of an equipment (see Figure 23). This differs from the <E139-JOB-1 – Process Job Setup> scenario in that the recipe is not allowed to remain persistent on the tool and must be removed immediately after it has been used (if it isn't to be used immediately for another queued lot). The typical expected usage scenario in this situation is that the FICS will send the relevant PDE to the equipment immediately before the corresponding process job is set up and delete the PDE immediately after the job has finished.



**Figure 23 E139-JOB-2 – Secure Recipe Use**

### 9.5.2.1 Pre-Conditions

The tool must be in a state ready to set up and run jobs. It is also assumed that the relevant recipe does not yet exist on the tool (although the sequence of messages should still function in that case).

### 9.5.2.2 Post-Conditions

The job is completed and the recipe is removed from the tools storage area again.

### 9.5.2.3 Notes

While the `requestToSendPDE()` message is technically optional, it is strongly recommended to use it in order to know whether the tool has enough space for the recipe before the transport begins.

Usually it should be possible to transport all PDEs of the required recipe in one step. However, if large PDEs are involved it may be necessary to repeat these steps several times in order to transport all PDEs.

The equipment is encouraged to verify the full PDE hierarchy (verifyDepth=All, verifyType=Checksum) used for a job before the recipe is used each time, unless it is confirmed that the equipment does this automatically.

The createJob call, JobStartedEvent and JobCompletedEvent are here for illustration, in practice they need to be replaced by whatever mechanism is used to set up and run jobs. Typically this will facilitate SEMI E40 and E94 Jobs.

The TraceabilityEvents encompass all events that inform the FICS about the processing state of the Job/PDE. These include events that tell the FICS about the start and end of process steps, as well as events informing the FICS about changes in PDE parameters.

#### 9.5.2.4 Exceptions

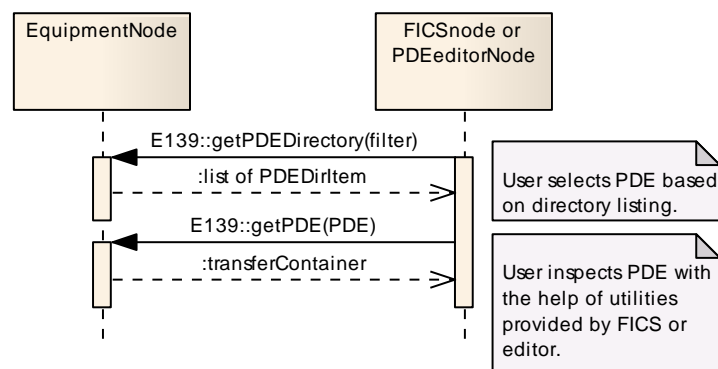
If the requestToSendPDE() or sendPDE() message fails (the response code is not “OK”) the scenario should be aborted and all PDEs that have already been transported should be deleted.

## 9.6 Troubleshooting and Maintenance Scenarios

### 9.6.1 <E139-MNT-1 – Inspect Recipe>

In the case of processing problems it may be desirable to check the recipe that was run for defects or potentially dangerous settings (see Figure 24). Apart from remotely verifying the recipe on the equipment, RaP provides the possibility to acquire the recipe from the equipment for inspection using factory tools.

Currently, the FICS may verify the checksum, review header information, and compare the PDE with the stored version. In next generation implementations it is anticipated the RMS will be able to provide more comprehensive inspection capabilities.



**Figure 24 E139-MNT-1 – Inspect Recipe**

If an issue is detected, the engineer or operator may choose to delete the offending recipe or download a new, correct copy from the FICS.

Typically a PDE Editor specialized for this equipment type, less frequently a generic PDE Editor or a FICS component with inspection capabilities will be used.

First the inspecting entity will request a list of PDEs from the equipment and give the user the possibility to choose a specific PDE, which will then be fetched using getPDE().

### 9.6.1.1 Pre-Conditions

A process problem has been detected.

### 9.6.1.2 Post-Conditions

The user has selected a PDE which then was transferred to the inspecting entity.

### 9.6.1.3 Notes

In the case of an advanced FICS with inspection capabilities the `getPDEdirectory()` call may be unnecessary if the FICS allows selection through what it already knows about the process run that caused the problems.

### 9.6.1.4 Exceptions

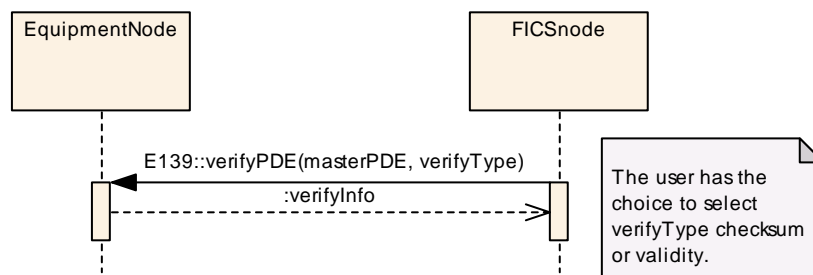
Any call in this scenario can fail. In this case it should simply be aborted. The user can then either retry or do the inspection directly at the equipment.

## 9.6.2 <E139-MNT-2 – Validate Recipe>

In the case of processing problems or as a periodic check, it may be desirable to validate a recipe that is resident on the equipment (see Figure 25). This scenario remotely requests the equipment to do a check of the recipe.

### 9.6.2.1 Pre-Conditions

The FICS has determined that validation of a recipe is appropriate. This may be due to a process problem involving that recipe.



**Figure 25 E139-MNT-2 – Validate Recipe**

### 9.6.2.2 Post-Conditions

The user has selected a PDE which then was checked at the equipment. The status of the check has been sent to the FICS in the response to the `verifyPDE()` call.

### 9.6.2.3 Notes

This scenario assumes that the FICS knows which recipe is to be validated. It is possible that the FICS may use the `getPDEdirectory()` service to ensure the recipe is present or allow a user to choose.

The `verifyPDE()` service can be used to check only the PDE referenced in the call (`depth=single`) or the whole hierarchy of PDEs referenced from it (`depth=all`). In this case it is recommended to do a full check.

The verifyPDE() service can be used to check only the checksum (verifyType=checksum) or also the syntax and validity of the entire recipe (verifyType=validity). If hard disk problems are suspected a checksum check may be sufficient (most data corruption will become visible through checksum checks).

#### 9.6.2.4 Exceptions

Any call in this scenario can fail. In this case it should be aborted. The user can then either retry or perform the inspection directly on the equipment.

### 9.6.3 <E139-MNT-3 – Software Update>

Updates to equipment software by the supplier and their effects on equipment recipes pose a significant challenge to the factory. New software versions may result in changes to the required format of the recipe, invalidating some or all of the existing recipes. It may also result in changes to expected equipment results based on use of the same or essentially similar recipes. Whenever a software update is detected, great care must be taken to ensure that processing is not compromised.

The actions that must be taken following a software update vary from supplier to supplier and even event-to-event. No fixed sequence of messages can be supplied to address this situation. Instead, this section will discuss the factors that must be considered with a software update occurs and suggest specific actions in certain situations.

#### 9.6.3.1 Recipe Conversion

The first question to be asked is whether the existing recipes can still be used on the equipment. This is first a question for the supplier to answer, but it can be confirmed by

- Downloading representative recipes to the equipment. Upon download, each PDE is verified automatically (verifyType = Validity). Any compatibility problems must be reported by the equipment.
- Verifying the downloaded recipes. Verification with verifyDepth=all will check the entire recipe structure to ensure it will work as a unit.

If the recipes are no longer compatible with the equipment, they must be converted to the new format. It is critical that the supplier provide a conversion program for this process. The conversion program should run with minimal user input and guide the user's selections when input is required.

It is expected that most software updates will not significantly affect the equipment operation. If, for example, the settings for a process module significantly change, then all PDEs for that module may have to be completely reconstructed. This is to be avoided if possible.

Note that a converted RaP recipe must have a different uid than the original. This is a challenge, but also an important protection. In older systems, the converted recipe may be given the same identifier as the old version. The old and new version are difficult to differentiate and problems may result in the old one is inadvertently used. This is not a problem with RaP PDEs.

The challenge is to provide support for the changeover of the factory recipe management system to use the converted PDEs. To ease this transition, it is strongly recommended that the conversion process place the original uid of the original PDE into the supplierInfo field in the

PDEheader of the converted PDE. The factory RMS can use this field to update itself automatically to use these converted PDEs in place of the old ones.

It is also recommended that device makers use group references in multi-part recipes. These “indirect” references allow for newer versions of sub-components of the recipe without changing the referencing PDE (for example, one can update a chamber PDE without modifying the sequence PDE that references it).

With this approach, the conversion process can be straightforward. It is recommended that the device maker eliminate all unneeded recipes before the software update begins to reduce the overall effort needed for the conversion.

### 9.6.3.2 Software Update Evaluation

It is critical that the user determine any changes to the expected results of a recipe based on this software update. The user should examine any supplier documentation of the software modifications represented in this update. It may be appropriate to re-qualify some or all of the recipes depending on the level of change to the software.

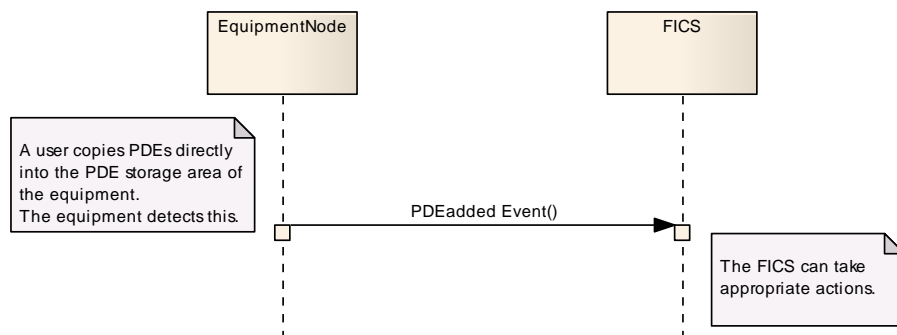
### 9.6.3.3 Synchronization

The recipe conversion process may occur at the equipment or with a separate application off the equipment. Once all the recipes have been converted, they must be introduced to both the factory RMS and to the equipment. This may entail recipe upload, download, or both. Any copies of the old version of the recipes should be deleted. The <E139-RMS-9 – Synchronize RMS and Equipment> scenario should be a helpful guide to this process and should be performed in total before the equipment is returned to production.

### 9.6.4 <E139-MNT-4 – Copy Recipe from Removable Media directly onto Tool>

Regardless of how recipes/PDEs end up in or are removed from the recipe storage area of an equipment, the equipment must send PDEadded/PDEremoved events to the FICS (see Figure 26). For example, PDEs may be transferred as files using removable media (like CDs or USB sticks) or via network file transfer.

It is therefore recommended that an equipment node monitor the recipe storage area and record changes to it accordingly.



**Figure 26 E139-MNT-4 Copy Recipe from Removable Media Directly Onto Tool**

#### **9.6.4.1 Pre-Conditions**

A user copies one or more PDEs to the storage area of the equipment.

#### **9.6.4.2 Post-Conditions**

The FICS will be informed about the new PDEs and can make decisions based on that knowledge. Note that the PDEs may already be known to the FICS.

#### **9.6.4.3 Notes**

What actions are taken depends on the specific scenario of the FICS. This could, for example, result in a backup of the new PDE to the FICS.

#### **9.6.4.4 Exceptions**

The PDEadded event is not received if the connection is offline or lost during the copying.

## **10 OVERVIEW OVER SCENARIOS**

Table 2 contains a list of all scenarios, with references to the section they are described and which diagrams are included with each. The table also provides for each scenario a brief description, a list of the messages used (for the Unit Scenarios only), and a column indicating whether those scenarios are used in either the Early or Next Generation approach (as described in Section 8).

**Table 2 Scenario Summary**

Section	Name	Diagram	Description	Messages Used	Applies To	
					Early Impl.	Next Gen
<b>Fundamental Scenarios</b>						
9.3.1	getPDEdirectory()	Figure 5	Getting a list of PDEs from a RaPnode.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.3.2	deletePDE()	Figure 6	Deleting a PDE.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.3.3	getPDEheader()	Figure 7	Getting header information about a PDE.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.3.4	getPDE()	Figure 8	Getting a PDE.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.3.5	requestToSendPDE()	Figure 9	Requesting permission to send a PDE to a RaPnode.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.3.6	sendPDE()	Figure 10	Sending a PDE to a RaPnode.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.3.7	resolvePDE()	Figure 11	Resolving links in a hierarchy of PDEs.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.3.8	verifyPDE()	Figure 12	Verifying a PDE or recipe.		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<b>Unit Scenarios</b>						
9.4.1	<E139-RMS-1 – Transfer Recipe to RMS (initiated by RMS)>	Figure 13	Transferring a recipe to the FICS/RMS. Initiated by the FICS.	getPDEdirectory() resolvePDE() getPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.4.2	<E139-RMS-2 – Transfer Recipe to RMS (from Equipment)>	Figure 14	Transferring a recipe to the FICS/RMS. Initiated by the operator.	requestToSendPDE() sendPDE()	<input checked="" type="checkbox"/>	<input type="checkbox"/>
9.4.3	<E139-RMS-3 – Non-Uploadable PDE Handling>	Figure 15	Transferring information about a recipe to the FICS/RMS if the recipe body is too big.	getPDEdirectory() resolvePDE() getPDEheader()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.4.4	<E139-RMS-4 – Equipment Recipe Store Change>	Figure 16	Notification about changes in the PDE directory.	PDEadded event PDEdeleted event	<input type="checkbox"/>	<input checked="" type="checkbox"/>
9.4.5	<E139-RMS-5 – Recipe Download>	Figure 17	Downloading a recipe.	requestToSendPDE() sendPDE() verifyPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.4.6	<E139-RMS-6 – PDE Superseded>	Figure 18	Replacing a recipe with a new version.	deletePDE() requestToSendPDE() sendPDE()	<input type="checkbox"/>	<input checked="" type="checkbox"/>
9.4.7	<E139-RMS-7 – Delete Recipe>	Figure 19	Deleting a PDE/recipe from the equipment.	deletePDE()	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Section	Name	Diagram	Description	Messages Used	Applies To	
					Early Impl.	Next Gen
9.4.8	<E139-RMS-8 – Edit Recipe>	Figure 20	Using a PDE editor that is external to the equipment.	getPDEdirectory() resolvePDE() getPDE() requestToSendPDE() sendPDE()	<input type="checkbox"/>	<input checked="" type="checkbox"/>
9.4.9	<E139-RMS-9 – Synchronize RMS and Equipment>	Figure 21	Ensuring the equipment has the same list of recipes that the RMS expects it to have.	getPDEdirectory() requestToSendPDE() sendPDE() deletePDE() getPDE()	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<b>Processing Scenarios</b>						
9.5.1	<E139-JOB-1 – Process Job Setup>	Figure 22	Running jobs in a factory environment with RaP.	resolvePDE() requestToSendPDE() sendPDE() verifyPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.5.2	<E139-JOB-2 – Secure Recipe Use>	Figure 23	Running jobs in a factory environment with Rap and with additional requirements of secrecy.	requestToSendPDE() sendPDE() verifyPDE() deletePDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<b>Troubleshooting and Maintenance Scenarios</b>						
9.6.1	<E139-MNT-1 – Inspect Recipe>	Figure 24	Inspecting a recipe for trouble shooting.	getPDEdirectory() getPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.6.2	<E139-MNT-2 – Validate Recipe>	Figure 25	Validating a recipe for trouble shooting.	verifyPDE()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.6.3	<E139-MNT-3 – Software Update>	<None>	Synchronization and validation after software changes.	<None>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9.6.4	<E139-MNT-4 – Copy Recipe from Removable Media directly onto Tool>	Figure 26	Handling of introduction of new PDEs or recipes directly on the tool.	PDEadded event	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>





**International SEMATECH Manufacturing Initiative  
Technology Transfer  
2706 Montopolis Drive  
Austin, TX 78741**

**<http://ismi.sematech.org>  
e-mail: [info@sematech.org](mailto:info@sematech.org)**