INTERNATIONAL STANDARD

ISO/IEC 23004-7

First edition
2008-02-15

Information technology — Multimedia Middleware —

Part 7: System integrity management

Technologies de l'information — Intergiciel multimédia —

Partie 7: Gestion de l'intégrité de système

Reference number
ISO/IEC 23004-6:2008(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 23004 may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 23004-7 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 23004 consists of the following parts, under the general title *Information technology — Multimedia Middleware*:

— *Part 1: Architecture*

— *Part 2: Multimedia application programming interface (API)*

— *Part 3: Component model*

— *Part 4: Resource and quality management*

— *Part 5: Component download*

— *Part 6: Fault management*

— *Part 7: System integrity management*

Introduction

Software systems continuously evolve; both during their development and during deployment. To cater for an increasing demand for flexibility, the M3W architecture provides means for the dynamic replacement, removal and addition of components in a deployed system. This facility enables a number of new usage scenarios which can increase the value of a terminal. However, this facility also endangers the software integrity of the system and creates the need for terminal management activities that aim to maintain and (if necessary) restore software integrity. Ideally, a terminal is managed automatically, implying little or no human intervention.

System Integrity management techniques aim to maintain/restore a consistent software configuration that is sufficient for a specific terminal. These techniques verify that component configurations "fit" with a device, where the following fits have been identified.

— Business fit: This depends for example on whether or not a user has paid for certain components.

— Technical fit: This deals with whether or not components are for the "right" platform, all required dependencies can be fulfilled, and design guidelines are obeyed.

— Resource fit: This deals with verifying that the resource demands of applications are in line with the capabilities of the terminal.

Information technology — Multimedia Middleware —

Part 7: System integrity management

1 Scope

This part of ISO/IEC 23004 defines the MPEG Multimedia Middleware (M3W) technology Integrity Management Architecture. It contains the specification of the part of the M3W application programming interface (API) related to Integrity Management as well as the realization. The M3W API specification provides a uniform view of the Integrity Management functionality provided by M3W. The specification of the realization is relevant for those who are making an implementation of an Integrity Management framework for M3W.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 23004-1, *Information technology — Multimedia Middleware — Part 1: Architecture*

ISO/IEC 23004-3, *Information technology — Multimedia Middleware — Part 3: Component model*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1 Specification terms and definitions

3.1.1
API specification
specification of a collection of software interfaces providing access to coherent streaming-related functionality

3.1.2
interface suite
collection of mutually related interfaces providing access to coherent functionality

3.1.3
logical component
coherent unit of functionality that interacts with its environment through explicit interfaces only

3.1.4
role
abstract class defining behavior only

3.1.5
role instance
object displaying the behavior defined by the role

3.1.6
attribute
instance variable associated with a role

NOTE Attributes are used to associate state information with roles.

3.1.7
signature
definition of the syntactic structure of a specification item such as a type, interface or function in IDL

NOTE For C functions, signature is equivalent to prototype.

3.1.8
specification item
entity defined in a specification

NOTE Data type, role, attribute, interface and function are examples of specification items.

3.1.9
qualifier
predefined keyword representing a property or constraint imposed on a specification item

3.1.10
constraint
restriction that applies to a specification item

3.1.11
execution constraint
constraint on multi-threaded behavior

3.1.12
model type
data type used for specification (modeling) purposes only

NOTE Set, map and entity are example of model types.

3.1.13
model constant
constant used for specification (modeling) purposes only

3.1.14
enum element type
enumerated type whose values can be used to construct sets (bit vectors) of at most 32 values by logical or-ing

3.1.15
enum set type
32-bit integer data type representing sets of enumerated values

3.1.16
set type
data type whose values are mathematical sets of values of a specific type

NOTE Unlike enum sets, these sets may be infinite.

3.1.17
map type
data type whose values are tables mapping values of one type (the domain type) to values of another type (the range type)

NOTE Maps are a kind of generalized array.

3.1.18
entity type
class of objects that may have attributes associated with them

3.1.19
interface-role model
extended Unified Modeling Language class diagram showing the roles and interfaces associated with a logical component, and their mutual relations

3.1.20
logical component instance
incarnation of a logical component: a configuration of objects displaying the behavior defined by the logical component

3.1.21
provides interface
interface that is provided by a role or role instance

3.1.22
requires interface
interface that is used by a role or role instance

3.1.23
specialization
behavoral inheritance
definition, by a role, of behavior which implies the behavior defined by another role

NOTE A role S specializes a role R if the behavior defined by S implies the behavior defined by R, i.e. if S has more specific behavior than R.

3.1.24
diversity
set of all parameters that can be set at instantiation time of a logical component, and that will not change during the lifetime of the logical component

3.1.25
mandatory interface
provides interface of a role that should be implemented by each instance of the role

3.1.26
optional interface
provides interface of a role that need not be implemented by each instance of the role

3.1.27
configurable item
parameter that can be set at instantiation time of a logical component, usually represented by a role attribute

3.1.28
diversity attribute
role attribute that represents a configurable item

3.1.29
instantiation
process of creating an instance (an incarnation) of a role or logical component

3.1.30
initial state
state of a role instance or logical component instance immediately after its instantiation

3.1.31
observable behavior
behavior that can be observed at the external software and streaming interfaces of a logical component

3.1.32
function behavior
behavior of the functions in the provides interfaces of a role

3.1.33
streaming behavior
input-output behavior of the streams associated with a role

3.1.34
active behavior
autonomous behavior that is visible at the provides and requires interfaces of a role

3.1.35
instantiation behavior
behavior of a role at instantiation time of a logical component

3.1.36
independent attribute
attribute whose value may be defined or changed independently of other attributes and entities

3.1.37
dependent attribute
attribute whose value is a function of the values of other attributes or entities

3.1.38
invariant
assertion about a role or logical component that is always true from an external observer's point of view

NOTE In reality, the assertion may temporarily be violated.

3.1.39
callback interface
interface provided by a client of a logical component whose functions are called by the logical component

NOTE A notification interface is an example of this, but there may be other call-back interfaces as well, e.g. associated with plug-ins.

3.1.40
callback-compliance
general constraint that the functions in a callback interface should not interfere with the behavior of the caller in an undesirable way, such as by blocking the caller or by delaying it too long

3.1.41
event notification
act of reporting the occurrence of event to 'interested' objects

3.1.42
event subscription
act of recording the types of event that should be notified to objects

3.1.43
cookie
special integer value that is used to identify an event subscription

NOTE Clients pass cookies to a logical component when subscribing to events. Logical components pass cookies back to clients when notifying the occurrence of the events.

3.1.44
event-action table
table associating events that can occur to actions that will be performed in reaction to the events

NOTE This is used to specify event-driven behavior.

3.1.45
non-standard event notification
event notification that is accompanied by other actions (such as state changes of the notifying logical component

3.1.46
client role
role modeling the users of a logical component

3.1.47
actor role
role (usually a client role) whose active behavior consists of calling functions in interfaces without any a priori constraints on when these calls will occur

3.1.48
control interface
interface provided by a logical component that allows the logical component's functionality to be controlled by a client

3.1.49
notification interface
interface provided by a client of a logical component that is used by the logical component to report the occurrence of events to the client

3.1.50
specialized interface
interface of a role R that is inherited from another role and is further constrained by R

3.1.51
precondition
assertion that should be true immediately before a function is called

3.1.52
action clause
part of an extended precondition and postcondition specification defining the abstract action performed by a function

NOTE The abstract action usually defines which variables are modified and/or which out-calls are made by the function.

3.1.53
out-call
An out-going function call of an object on an interface of another object

3.1.54
postcondition
assertion that will be true immediately after a function has been called

3.1.55
asynchronous function
function with a delayed effect

NOTE The effect of the function will occur some time after returning from the function call.

3.2 Realization terms and definitions

3.2.1
application layer
software Layer that contains the software entities that provide functions to a user

3.2.2
middleware layer
M3W and other middleware

3.2.3
platform layer
operating system (OS) and hardware that executes the OS

3.2.4
error
unwanted software state that is liable to lead to a failure

3.2.5
failure
event that occurs when the delivered service of a system deviates from correct service

3.2.6
fault
adjudged or hypothesized cause of an error

3.2.7
monitoring
retrieving the model describing the current configuration of a terminal

3.2.8
diagnosis
identification of faults in the configuration of the terminal based on the model that describes the configuration

3.2.9
repair
removal of faults from the software configuration of the terminal

4 Abbreviated terms

API	Application Programming Interface
IDL	Interface Definition Language
M3W	Multimedia Middleware
OS	Operating System
UML	Unified Modeling Language

5 Overview of interface suites

This is an informative clause. The M3W Integrity Management framework enables maintaining and restoring a consistent software configuration on a terminal in the period that it is owned and used by a user. The API for Integrity Management is specified using four logical components:

— Terminal: The main responsibilities of the Terminal are to expose a model of the current software configuration and to offer basic software configuration facilities.

— Terminal Manager: The main responsibilities of the Terminal Manager are monitoring (model retrieving), diagnosis (finding faults in the software configuration based) and repair (generation and execution of a repair script that consists of basic configuration actions).

— Database: The main responsibility of the Database is to provide knowledge. Knowledge is used for diagnosis (rules) and for repair (solutions).

— Notification: This logical component enables triggering the Integrity Management framework when a lot of errors occur. The notification logical component will coordinate the monitor, diagnosis and repair actions of the terminal manager. This notification can be used by a Fault Management framework as specified in ISO/IEC 23004-6.

Integration of the Integrity Management Framework shall be done according to ISO/IEC 23004-1. The Integrity Management framework uses the Component Model as specified in ISO/IEC 23004-3.

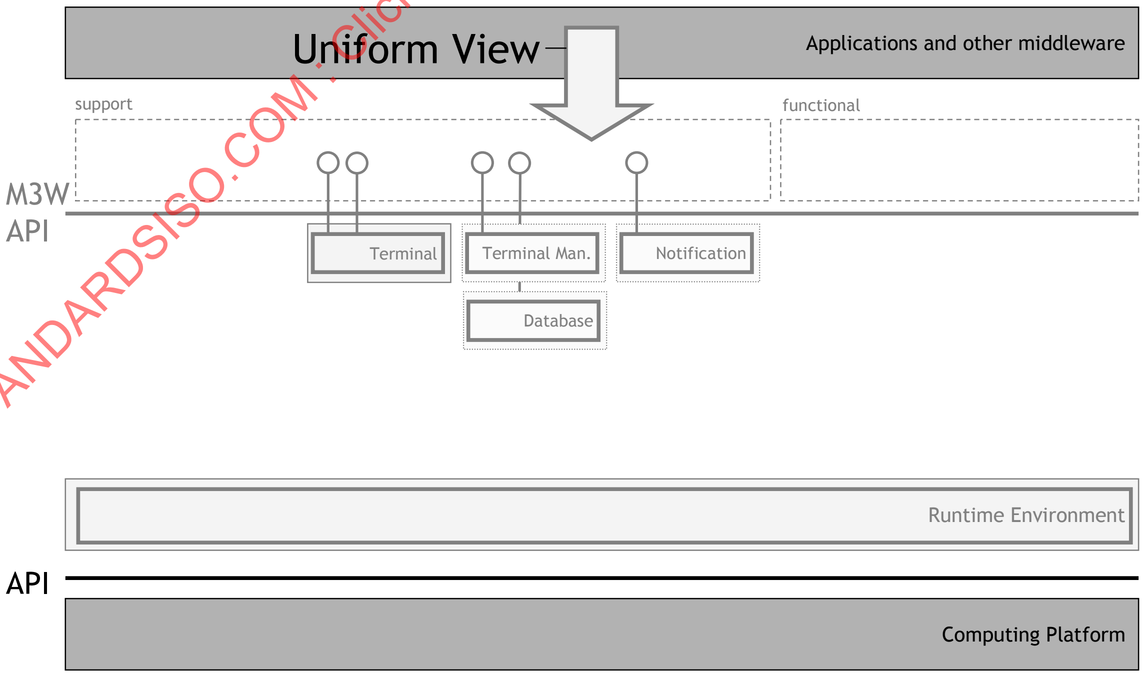

Figure 1 — M3W Integrity Management API

6 Integrity Management interface suites

6.1 Interface suites for extraction and modification of the software configuration

6.1.1 Terminal

6.1.1.1 Concepts

The Integrity Management framework enables maintaining a consistent software configuration of a device that can be upgraded and extended in the period that it is owned and used by a consumer. One of the roles in the Integrity Management framework is the terminal role. This role has two major responsibilities:

— Enable extraction of a model of the current software configuration: This model contains information like the executable components that are registered, services that are registered, complies relations that are registered, applications that are installed, etc.

— Provide basic configuration facilities: This includes facilities for (un)registration of executable components with the runtime environment, (un)installation of models, execution of models and downloading of new M3W components.

The terminal role is usually used by the Terminal Manager role, which is responsible for monitoring, diagnosis and repair. The Terminal provides the Terminal Manager with a means to extract information and to apply changes.

6.1.1.2 Types and Constants

6.1.1.2.1 Public Types and Constants

6.1.1.2.1.1 Error Codes

Signature

```
const rcResult RC_ERR_TERMINAL_NOT_ALLOWED = 0x00000001;
const rcResult RC_ERR_TERMINAL_INVALID_LOCATION = 0x00000002
const rcResult RC_ERR_TERMINAL_INVALID_COMPONENT = 0x00000004
const rcResult RC_ERR_TERMINAL_ALREADY_REGISTERED = 0x00000008
const rcResult RC_ERR_TERMINAL_UNKNOWN_COMPONENT = 0x00000010
```

Qualifiers

— Error-codes

Description

The non-standard error codes that can be returned by functions of this logical component

Constants

Table 1

Name	Description
RC_ERR_TERMINAL_NOT_ALLOWED	This error is returned when the operation is not allowed
RC_ERR_TERMINAL_INVALID_LOCATION	This error is returned on an attempt to use an invalid location. For example upon registration of an executable component.
RC_ERR_TERMINAL_INVALID_COMPONENT	This error is returned on an attempt to register an invalid executable component.
RC_ERR_TERMINAL_ALREADY_REGISTERED	This error is returned on an attempt to register an executable component multiple times.
RC_ERR_TERMINAL_UNKNOWN_COMPONENT	This error is returned on an attempt to modify the registered information related to an executable component that is not known to the runtime environment.

6.1.1.2.1.2 rcRuntime_ComponentRecord_t

Signature

```
struct _rcRuntime_ComponentRecord_t {
           UUID cmpId;
           String location;
} rcRuntime_ComponentRecord_t, *prcRuntime_ComponentRecord_t;
```

Qualifiers

— struct-element

Description

Data structure used to pass and store registration of an executable component.

6.1.1.2.1.3 rcRuntime_ContainerRecord_t

Signature

```
struct _rcRuntime_ContainerRecord_t {
           UUID cmpId;
           UUID svcId;
} rcRuntime_ContainerRecord_t, *prcRuntime_ContainerRecord_t;
```

Qualifiers

— struct-element

Description

Data structure used to pass and store registration of the containment of a service by an executable component.

6.1.1.2.1.4 rcRuntime_CompliesRecord_t

Signature

```
struct _rcRuntime_CompliesRecord_t {
           UUID complying;
           UUID blueprint;
} rcRuntime_CompliesRecord_t, *prcRuntime_CompliesRecord_t;
```

Qualifiers

— struct-element

Description

Data structure used to pass and store registration of the complies relation between services.

6.1.1.2.1.5 rcTerminal_ModelRecord_t

Signature

```
struct _rcTerminal_ModelRecord_t {
           UUID mdlId;
           String location;
} rcTerminal_ModelRecord_t, *prcTerminal_ModelRecord_t;
```

Qualifiers

— struct-element

Description

Data structure used to model the installed models.

6.1.1.2.1.6 rcTerminal_M3WComponentRecord_t

Signature

```
struct _rcTerminal_M3WComponentRecord_t {
           UUID m3wCmpId;
           String location;
} rcTerminal_M3WComponentRecord_t, *prcTerminal_M3WComponentRecord_t;
```

Qualifiers

— struct-element

Description

Data structure used to model the (downloaded) M3W components that are available on the device.

6.1.1.2.2 Model Types and Constants

None

6.1.1.3 Logical Component

6.1.1.3.1 Interface Role Model

Figure 2 — Interface Role Model depicts the interface-role model of the Terminal logical component (grey box). It shows the interfaces and the roles involved with this component.

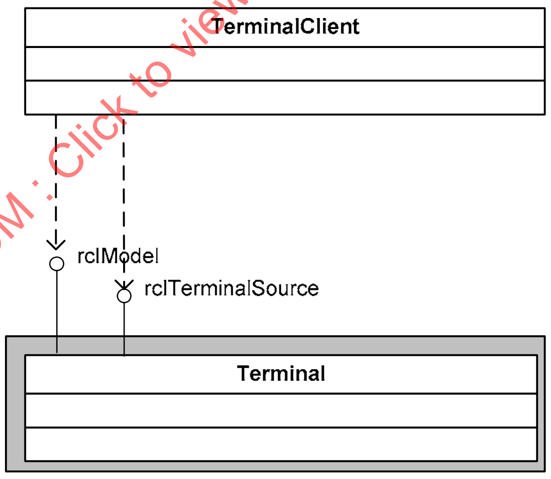

Figure 2 — Interface Role Model

The `Terminal` logical component contains the `Terminal` role. The Terminal role implements the `rcIModel` interface and the `rcITerminalSource` interface. The `rcIModel` interface is used to expose a model of the configuration of the terminal. The `rcITerminalSource` interface offers a number of basic configuration facilities to change the software configuration of the terminal.

6.1.1.3.2 Diversity

6.1.1.3.2.1 Provided Interfaces

Table 2

Role	**Interface**	**Presence**
`Terminal`	`rcIModel`	`Mandatory`
`Terminal`	`rcITerminalSource`	`Mandatory`

6.1.1.3.2.2 Configurable Items

The behaviour of the Runtime Environment depends on a registry that contains information on:

— the available executable components,

— which services are contained by these executable components, and

— which services are compatible.

The registry content can be changed using the basic configuration facilities of the terminal role. The registry content is specified using the following attributes: `componentRecords`, `containerRecords` and `compliesRecords`.

The Terminal role also supports downloading of M3W components and installation of models that are contained by these M3W components. The installed models are modeled by the `modelRecords`. The M3W Components that are resident on the device are models using the `m3wComponentRecords`

Table 3

Role	**Attribute**
Terminal	`componentRecords`
Terminal	`containerRecords`
Terminal	`compliesRecords`
Terminal	`modelRecords`
Terminal	`m3wComponentRecords`

6.1.1.3.2.3 Constraints

None.

6.1.1.3.3 Instantiation

6.1.1.3.3.1 Objects Created

The `Terminal` role is a singleton. At most one instance on a device can be created.

Table 4

Type	Object	Multiplicity
`Terminal`	`terminal`	`1`

6.1.1.3.3.2 Initial State

The following constraints apply to the initial state of a logical component instance:

— none

6.1.1.3.4 Execution Constraints

The `Terminal` logical component is thread-safe.

6.1.1.4 Roles

6.1.1.4.1 Terminal

Signature

```
role Terminal {
  rcRuntime_ComponentRecord_t       componentRecords[];
  rcRuntime_ContainerRecord_t       containerRecords[];
  rcRuntime_CompliesRecord_t        compliesRecords[];
  rcTerminal_ModelRecord_t          modelRecords[];
  rcTerminal_M3WComponentRecord_t   m3wComponentRecords[];
}
```

Qualifiers

— Root

Description

The Terminal role enables extraction of a model (`rcIModel`) of the current configuration of a device and provides a number of basic configuration facilities (`rcITerminalSource`) to modify this configuration. The model that can be extracted can be used for diagnostic purposes. The basic configuration facilities can be used to modify the configuration (repair).

Independent Attributes

Table 5

Attribute	Description
`componentRecords`	`Used to store the uuid's of the registered executable components and the location on locally accessible storage.`
`containerRecords`	`Used to store the binding between services and executable components.`
`compliesRecords`	`Used to store the "complies" relation between services.`
`modelRecords`	`Used to store the models that are installed on the device.`
`m3wComponentRecords`	`Used to store the M3W Components that are resident on the device.`

Invariants

— None

Instantiation

The `Terminal` role is always created as part of the `Terminal` logical component.

Active Behavior

The `Terminal` role has no active behavior.

6.1.1.5 Interfaces

6.1.1.5.1 rclModel

Qualifiers

— None

Description

This interface enables extraction (and some manipulation) of a model of the current configuration of the device. This model can be used for diagnostic purposes part of the Integrity Management framework.

Interface ID

`{ CE752B0D-3795-D811-87C6-0008744C31AC }`

6.1.1.5.1.1 inspect

Signature

```
rcResult inspect (
  [in] String elementId,
  [out] String element
);
```

Qualifiers

— Thread-safe

Description

Retrieve a selection of the model that describes the configuration of the device.

Parameters

Table 6

Name	Description
`elementId`	XPath [13] expression that identifies a selection of the model of the configuration of the device (self model).
`element`	Output parameter, used for returning an XML fragment that contains the requested selection.

Return Values

Standard.

Precondition

`ElementId = X`

`"Selfmodel" = M`

Action

None.

Postcondition

`Element = "XML fragment of the self model selected by X"`

`"Selfmodel" = M`

6.1.1.5.1.2 update

Signature

```
rcResult update (
  [in] String elementId,
  [in] String value
);
```

Qualifiers

— Thread-safe

Description

Update the model of the configuration (self model) of the device. This is used to modify the parts of the self model that are not generated (based on, for example, the registry contents)

Parameters

Table 7

Name	Description
`elementId`	XPath [13] expression that identifies a selection of the model of the configuration of the device (self model).
`value`	New value for the selection.

Return Values

Only non-standard error values are listed

Table 8

Name	Description
`RC_ERR_TERMINAL_NOT_ALLOWED`	This error is returned when the operation is not allowed.

Precondition

`ElementId = X`

Action

None.

Postcondition

```
X specifies part of self model that is generated automatically ⇒
   return value is RC_ERR_TERMINAL_NOT_ALLOWED
otherwise self model is updated according to value.
```

6.1.1.5.1.3 take

Signature

```
rcResult take (
  [in] String elementId,
  [out] String element
);
```

Qualifiers

— Thread-Safe

Description

Retrieve and remove a selection of the model of the configuration of the device (self model).

Parameters

Table 9

Name	Description
`elementId`	XPath [13] expression that identifies a selection of the model of the configuration of the device (self model)
`element`	Output parameter, used for returning an XML fragment that contains the requested selection.

Return Values

Only non-standard error values are listed

Table 10

Name	Description
`RC_ERR_TERMINAL_NOT_ALLOWED`	`This error is returned when the operation is not allowed`

Precondition

```
ElementId = X

"Selfmodel" = M
```

Action

None.

Postcondition

```
Element = "XML fragment of the self model selected by X"

"Selfmodel" = M \ "fragment of the self model selected by X"
```

6.1.1.5.2 rclTerminalSource

Qualifiers

— Thread-Safe

Description

This interface provides basic configuration facilities that enable modification of the configuration of the device. This includes changing the registry contents, downloading M3W components and installation of models contained by the M3W components.

Interface ID

```
{ 08D0F381-2099-D811-87C6-0008744C31AC }
```

6.1.1.5.2.1 registerComponent

Signature

```
rcResult registerComponent (
  [in] pUUID cmpId,
  [in] String location
);
```

Qualifiers

synchronous

thread-safe

Description

Add the executable component to the registry so that it can be loaded dynamically at a later time.

Parameters

Table 11

Name	Description
cmpId	Reference to the identified of the executable component.
location	Location on locally accessible storage where the executable component is located

Return Values

Only non-standard error values are listed

Table 12

Name	Description
RC_ERR_TERMINAL_INVALID_LOCATION	The location is not valid
RC_ERR_TERMINAL_INVALID_COMPONENT	The executable component is not valid (does not comply with the M3W component model).
RC_ERR_TERMINAL_ALREADY_REGISTERED	The component is already registered

Precondition

```
componentRecords = A
```

Action

None

Postcondition

```
componentRecords = A ∪ <cmpId, location>
```

6.1.1.5.2.2 unregisterComponent

Signature

```
rcResult unregisterComponent (
   [in] pUUID cmpId
);
```

Qualifiers

synchronous

thread-safe

Description

Remove the executable component from the registry.

Parameters

Table 13

Name	Description
`cmpId`	`Reference to the identifier of the executable component.`

Return Values

Default.

Precondition

```
componentRecords = A ∪ <cmpId, location>
```

Action

None

Postcondition

```
componentRecords = A
```

6.1.1.5.2.3 registerService

Signature

```
rcResult registerService (
  [in] pUUID cmpId,
  [in] pUUID svcId
);
```

Qualifiers

synchronous

thread-safe

Description

Add the service to the registry so that it can be instantiated at a later time. This operation creates the binding between the executable component and the service.

Parameters

Table 14

Name	Description
cmpId	Reference to the identifier of the executable component.
svcId	Reference to the identifier of the service

Return Values

Only non-standard error values are listed

Table 15

Name	Description
RC_ERR_TERMINAL_UNKNOWN_COMPONENT	The executable component is not registered yet.
RC_ERR_TERMINAL_ALREADY_REGISTERED	The service is already registered

Precondition

containerRecords = A

A ∩ < X, svcId> = ∉ for all X

Action

None

Postcondition

containerRecords = A ∪ <cmpId, svcId>

6.1.1.5.2.4 unregisterService

Signature

```
rcResult unregisterService (
  [in] pUUID svcId
);
```

Qualifiers

synchronous

thread-safe

Description

Remove the service from the registry. This operation removes the binding between the executable component and the service.

Parameters

Table 16

Name	Description
`svcId`	`Reference to the identifier of the service`

Return Values

Default.

Precondition

`containerRecords = A`

Action

None

Postcondition

`containerRecords = A`

`A ∩ < X, svcId> = ∉ for all X`

6.1.1.5.2.5 registerComplies

Signature

```
rcResult registerComplies (
   [in] pUUID complying,
   [in] pUUID blueprint
);
```

Qualifiers

synchronous

thread-safe

Description

Add a complies relation to the registry.

Parameters

Table 17

Name	Description
`complying`	`Reference to the identifier of the compliant service.`
`blueprint`	`Reference to the identifier of the blueprint service`

Return Values

Only non-standard error values are listed

Table 18

Name	Description
`RC_ERR_TERMINAL_NOT_ALLOWED`	`This operation is not allowed`

Precondition

`compliesRecords = A`

Action

None

Postcondition

`containerRecords = A ∪ <complying, blueprint>`

6.1.1.5.2.6 unregisterComplies

Signature

```
rcResult unregisterComplies (
  [in] pUUID complying
  [in] pUUID blueprint
);
```

Qualifiers

synchronous

thread-safe

Description

Remove a complies relation from the registry.

Parameters

Table 19

Name	**Description**
`complying`	`Reference to the identifier of the compliant service.`
`blueprint`	`Reference to the identifier of the blueprint service`

Return Values

Default

Precondition

```
compliesRecords = A
```

Action

None

Postcondition

```
containerRecords = A \ <complying, blueprint>
```

6.1.1.5.2.7 installModel

Signature

```
rcResult installModel (
  [in] pUUID m3wCmpId,
  [in] pUUID mdlId,
  [in] String location
);
```

Qualifiers

synchronous

thread-safe

Description

Install a model from a M3W Component on the device.

Parameters

Table 20

Name	Description
`m3wCmpId`	Reference to the UUID of the M3W Component that contains the model
`mdlId`	Reference to the UUID of the model that needs to be installed on the device
`location`	Location where to install the model

Return Values

Only non-standard error values are listed

Table 21

Name	Description
`RC_ERR_TERMINAL_INVALID_LOCATION`	`This error is returned on an attempt to use an invalid location. For example for registration of an executable component`
`RC_ERR_TERMINAL_NOT_ALLOWED`	`This operation is not allowed`

Precondition

`m3wCmpId ∈ m3wComponentRecords`

M3W Component with `m3wCmpId` contains model with `mdlId`

Action

Install model at the specified location

Postcondition

`<mdlId,location> ∉ modelRecords`

6.1.1.5.2.8 uninstallModel

Signature

```
rcResult uninstallModel (
  [in] pUUID mdlId
);
```

Qualifiers

synchronous

thread-safe

Description

Uninstall the specified model.

Parameters

Table 22

Name	Description
`mdlId`	Reference to the UUID of the model that needs to be uninstalled

Return Values

Only non-standard error values are listed

Table 23

Name	Description
`RC_ERR_TERMINAL_NOT_ALLOWED`	`This operation is not allowed`

Precondition

`<mdlId,location> ∊ modelRecords`

Action

Uninstall (delete) model

Postcondition

`<mdlId,location> ∉ modelRecords`

6.1.1.5.2.9 executeModel

Signature

```
rcResult executeModel (
  [in] pUUID mdlId
);
```

Qualifiers

synchronous

Description

Execute a model. This is only possible when the model is executable such as scripts.

Parameters

Table 24

Name	Description
`mdlId`	Reference to the UUID of the model that needs to be executed.

Return Values

Only non-standard error values are listed

Table 25

Name	Description
`RC_ERR_TERMINAL_NOT_ALLOWED`	`This operation is not allowed`

Precondition

```
modelRecord = M

<mdlId,location> ∊ M
```

Action

Execute the specified model

Postcondition

```
modelRecord = M
```

6.1.1.5.2.10 downloadM3WComponent

Signature

```
rcResult downloadM3WComponent (
   [in] pUUID m3wCmpId
);
```

Qualifiers

Synchronous

Thread-safe

Description

Download a new M3W component. After download the M3W component is resident on the device.

Parameters

Table 26

Name	Description
`m3wCmpId`	Reference to the UUID of the M3W component that needs to be downloaded.

Return Values

Default

Precondition

`true`

Action

None

Postcondition

`m3wCmpId ∊ m3wComponentRecords`

6.1.1.5.2.11 removeM3WComponent

Signature

```
rcResult removeM3WComponent (
   [in] pUUID m3wCmpId
);
```

Qualifiers

Synchronous

Thread-safe

Description

Remove a M3W Component from the device

Parameters

Table 27

Name	Description
`m3wCmpId`	Reference to the UUID of the M3W component that needs to be removed.

Return Values

Default

Precondition

True

Action

None

Postcondition

`m3wCmpId ∉ m3wComponentRecords`

6.2 Management activation interface suites

6.2.1 Notification

6.2.1.1 Concepts

The Integrity Management framework enables maintaining a consistent software configuration of a device that can be upgraded and extended in the period that it is owned and used by a consumer. One of the roles in the Integrity Management framework is the notification role. This role can be used to notify the Integrity Management framework that the device is not working properly. As a result, the Integrity Management Framework will monitor, diagnose and when necessary repair the configuration of the device on a separate thread.

6.2.1.2 Types and Constants

6.2.1.2.1 Public Types and Constants

6.2.1.2.1.1 Error Codes

Default

6.2.1.2.2 Model Types and Constants

None

6.2.1.3 Logical Component

6.2.1.3.1 Interface Role Model

Figure 3 — Interface Role Model depicts the interface-role model of the Notification logical component (grey box). It shows the interfaces and the roles involved with this component.

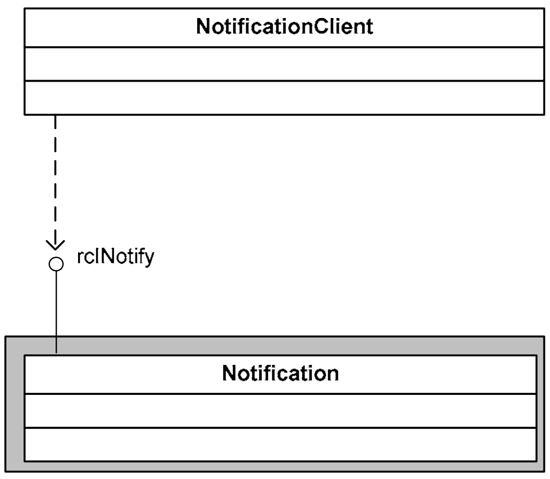

Figure 3 — Interface Role Model

The `Notification` logical component contains the `Notification` role. The `Notification` role implements the `rcINotify` interface. The `rcINotify` interface can be used to trigger system integrity management. Calling one of the operations will start monitoring, diagnosis and repair on a different thread of control.

6.2.1.3.2 Diversity

6.2.1.3.2.1 Provided Interfaces

Table 28

Role	Interface	Presence
`Notification`	`rcINotify`	`Mandatory`

6.2.1.3.2.2 Configurable Items

None

6.2.1.3.2.3 Constraints

None.

6.2.1.3.3 Instantiation

6.2.1.3.3.1 Objects Created

Table 29

Type	Object	Multiplicity
`Notification`	`notification`	`1`

6.2.1.3.3.2 Initial State

The following constraints apply to the initial state of a logical component instance:

— none

6.2.1.3.4 Execution Constraints

The `Notification` logical component is thread-safe.

6.2.1.4 Roles

6.2.1.4.1 Notification

Signature

```
role Notification {
}
```

Qualifiers

— Root

Description

The Notification role is used to notify the Integrity Management framework that the device is not working properly. As a result, the notification role coordinates monitoring, diagnosis and repair by the other roles of the Integrity Management Framework on a different thread of control.

Independent Attributes

None

Invariants

— None

Instantiation

The `Notification` role is always created as part of the `Notification` logical component.

Active Behavior

The `Notification` role has no active behavior.

6.2.1.5 Interfaces

6.2.1.5.1 rclNotify

Qualifiers

— None

Description

This interface contains two operations for notification to the integrity management framework that the device is not working properly. One is without additional information. The other one is provides the UUID of the service from which an instance is having errors.

Interface ID

```
{ 3eba62df-8ab8-4832-be31-40f871874eb3 }
```

6.2.1.5.1.1 notify

Signature

```
rcResult notify (
);
```

Qualifiers

- asynchronous
- thread-safe

Description

Notify Integrity Management framework that the device is not working properly without further information.

Parameters

None

Return Values

Standard

Precondition

`true`

Action

Coordinate monitor, diagnose and repair by the other roles of the Integrity Management Framework.

Postcondition

True

6.2.1.5.1.2 notifyWithServiceId

Signature

```
rcResult notifyWithServiceId (
  [in] pUUID svcId
);
```

Qualifiers

— asynchronous

— thread-safe

Description

Notify Integrity Management framework that the device is not working properly with information about the UUID of the service from which an instance is having errors.

Parameters

Table 30

Name	Description
`svcId`	Reference to the UUID of the service from which an instance is having errors.

Return Values

Default

Precondition

`true`

Action

Coordinate monitor, diagnose and repair by the other roles of the Integrity Management Framework.

Postcondition

`true`

6.2.2 Terminal Manager

6.2.2.1 Concepts

The Integrity Management framework enables maintaining a consistent software configuration of a device that can be upgraded and extended in the period that it is owned and used by a consumer. One of the roles in the Integrity Management framework is the Terminal Manager role. This role has 3 major responsibilities:

— Monitoring: This basically comes down to extraction of the model that describes the configuration of the terminal (self model).

— Diagnosis: Based on the self model the Terminal Manager will need to determine whether there is a fault in the configuration.

— Repair: The faults that have been identified during diagnosis will need to be removed when this is possible. The Terminal Manager will remove the faults by modifying the configuration using the basic configuration facilities offered by the Terminal role.

6.2.2.2 Types and Constants

6.2.2.2.1 Public Types and Constants

6.2.2.2.1.1 Error Codes

Default

6.2.2.2.2 Model Types and Constants

None

6.2.2.3 Logical Component

6.2.2.3.1 Interface Role Model

Figure 4 — Interface Role Model depicts the interface-role model of the TerminalManager logical component (grey box). It shows the interfaces and the roles involved with this component.

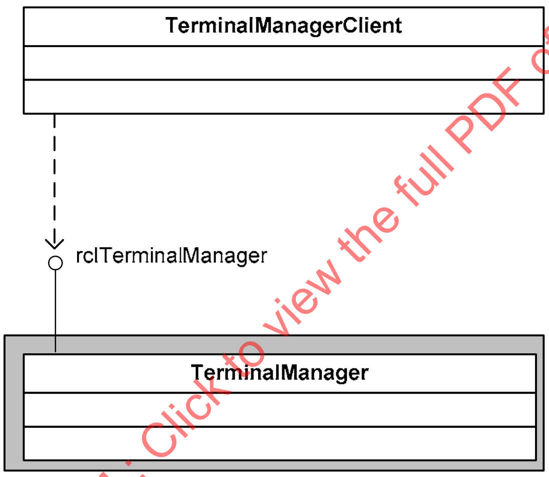

Figure 4 — Interface Role Model

The `TerminalManager` logical component contains the `TerminalManager` role. The `TerminalManager` role implements the `rcITerminalManager` interface. The `rcITerminalManager` interface can be used to start monitoring (retrieve model of the configuration), diagnosis (find faults in the configuration) and repair (remove faults from the configuration).

6.2.2.3.2 Diversity

6.2.2.3.2.1 Provided Interfaces

Table 31

Role	**Interface**	**Presence**
`TerminalManager`	`rcITerminalManager`	`Mandatory`

6.2.2.3.2.2 Configurable Items

None

6.2.2.3.2.3 Constraints

None

6.2.2.3.3 Instantiation

6.2.2.3.3.1 Objects Created

Table 32

Type	Object	Multiplicity
`TerminalManager`	`terminalManager`	1

6.2.2.3.3.2 Initial State

The following constraints apply to the initial state of a logical component instance:

— none

6.2.2.3.4 Execution Constraints

The `TerminalManager` logical component is thread-safe.

6.2.2.4 Roles

6.2.2.4.1 TerminalManager

Signature

```
role TerminalManager {
}
```

Qualifiers

— Root

Description

The TerminalManager role is responsible for monitoring, diagnosis and repair. This role usually uses knowledge from a database (rules and solutions) and the facilities provided by the terminal role (extraction of the self model and basic configuration facilities).

Independent Attributes

None

Invariants

— None

Instantiation

The `TerminalManager` role is always created as part of the `TerminalManager` logical component.

Active Behavior

The `TerminalManager` role has no active behavior.

6.2.2.5 Interfaces

6.2.2.5.1 rcITerminalManager

Qualifiers

— None

Description

This is the interface of the TerminalManager that contains the operations for monitoring, diagnosis and repair.

Interface ID

```
{ 04ABEB2C-3795-D811-87C6-0008744C31AC }
```

6.2.2.5.1.1 monitor

Signature

```
rcResult monitor (
  [out] String model,
  [out] Bool possibleProblem
);
```

Qualifiers

— synchronous

Description

Extraction of the model of the current configuration (self model) of the device and initial indication on whether there are possible problems.

Parameters

Table 33

Name	**Description**
`model`	Output parameter used to return the self model (xml fragment)
`possibleProblem`	Output parameter used to indicate whether there is a possible problem (for example because it changed since the last diagnosis)

Return Values

Standard

Precondition

`true`

Action

None

Postcondition

model = "XML fragment describing the configuration of the device"

possibleProblem = "First indication on whether there are possible faults in the configuration"

More detailed specification of the pre and post condition is platform specific and out of scope of this standard

6.2.2.5.1.2 diagnose

Signature

```
rcResult diagnose (
  [in] String model,
  [out] String diagnosis
);
```

Qualifiers

— synchronous

Description

Diagnosis of the configuration based on the self model.

Parameters

Table 34

Name	Description
`model`	XML fragment that describes the current configuration of the device (self model)
`diagnosis`	XML fragment that contains a description of the faults in the configuration.

Return Values

Default

Precondition

```
model = M
```

Action

None

Postcondition

```
diagnosis = "Faults in the configuration of the device based on M"
```

More detailed specification of the pre and post condition is platform specific and out of scope of this standard.

6.2.2.5.1.3 repair

Signature

```
rcResult repair (
   [in] String model,
   [in] String diagnosis,
   [out] Int32 remainingFaults
);
```

Qualifiers

— synchronous

Description

Repair the faults that have been identified during diagnosis.

Parameters

Table 35

Name	**Description**
`model`	XML fragment that describes the current configuration of the device (self model)
`diagnosis`	XML fragment that contains a description of the faults in the configuration.
`remainingFaults`	Output parameter that contains the number of faults that could not be removed.

Return Values

Default

Precondition

`model = M`

`diagnosis = D`

Action

Remove the faults expressed in D.

Postcondition

`remainingFaults = number of faults that could not be removed.`

More detailed specification of the pre and post condition is platform specific and out of scope of this part of ISO/IEC 23004.

6.3 Support interface suites

6.3.1 Database

6.3.1.1 Concepts

The Integrity Management framework enables maintaining a consistent software configuration of a device that can be upgraded and extended in the period that it is owned and used by a consumer. One of the roles in the Integrity Management framework is the Database role. The database provides the rules and solutions that are used by the integrity management framework. Rules are used during diagnosis, where a number of checks are performed in order to find the rules that are violated. Known solutions are used for repairing the configuration of a device.

6.3.1.2 Types and Constants

6.3.1.2.1 Public Types and Constants

6.3.1.2.1.1 Error Codes

Signature

```
const rcResult RC_ERR_DATABASE_NOT_ALLOWED = 0x00000001;
```

Qualifiers

— Error-codes

Description

The non-standard error codes that can be returned by functions of this logical component.

Constants

None

6.3.1.2.2 Model Types and Constants

None

6.3.1.3 Logical Component

6.3.1.3.1 Interface Role Model

Figure 5 — Interface Role Model depicts the interface-role model of the Database logical component (grey box). It shows the interfaces and the roles involved with this component.

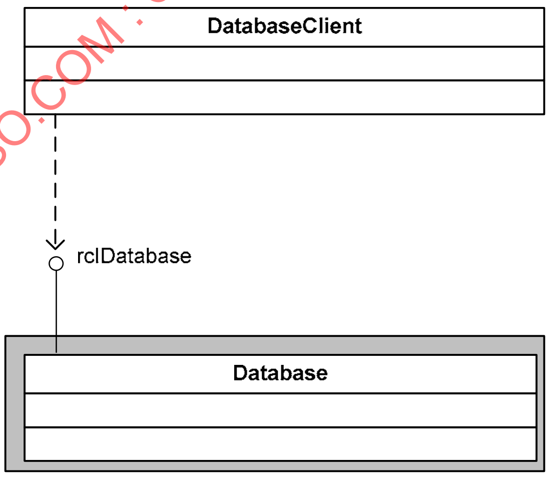

Figure 5 — Interface Role Model

The `Database` logical component contains the `Database` role. The `Database` role implements the `rcIDatabase` interface. The `rcIDatabase` interface provides some basic database operations that enable accessing the rules and solutions used for System Integrity Management.

6.3.1.3.2 Diversity

6.3.1.3.2.1 Provided Interfaces

Table 36

Role	Interface	Presence
`Database`	`rcIDatabase`	`Mandatory`

6.3.1.3.2.2 Configurable Items

None

6.3.1.3.2.3 Constraints

None

6.3.1.3.3 Instantiation

6.3.1.3.3.1 Objects Created

Table 37

Type	Object	Multiplicity
`Database`	`database`	`1`

6.3.1.3.3.2 Initial State

The following constraints apply to the initial state of a logical component instance:

— none

6.3.1.3.4 Execution Constraints

The `Database` logical component is thread-safe.

6.3.1.4 Roles

6.3.1.4.1 Database

Signature

```
role Database {
}
```

Qualifiers

— Root

Description

The database role provides the knowledge for the Integrity Management Framework. This includes rules that can be checked during diagnosis of a device as well as solutions for faults that have been found.

Independent Attributes

None

Invariants

— None

Instantiation

The `Database` role is always created as part of the `Database` logical component.

Active Behavior

The `Database` role has no active behavior.

6.3.1.5 Interfaces

6.3.1.5.1 rclDatabase

Qualifiers

— None

Description

The rclDatabase interface offers basic operations for accessing a database. The structure of these database (tables) is platform specific and out of the scope of this specification.

Interface ID

```
{ E421F248-1E99-D811-87C6-000EA69DB1A4 }
```

6.3.1.5.1.1 open

Signature

```
rcResult open (
  [out] pVoid connection
);
```

Qualifiers

— synchronous

— thread-safe

Description

Open a new database connection

Parameters

Table 38

Name	Description
`connection`	Output parameter that contains a pointer to the connection

Return Values

Standard

Precondition

`true`

Action

None

Postcondition

```
connection = pointer to connection
```

`connection` is of type `pVoid` the structure of the connection will depend on the database implementation used and is out of scope of this specification. The client shall not use the `connection` itself. It shall only pass it to the other operations of `rcIDatabase`.

6.3.1.5.1.2 close

Signature

```
rcResult close (
  [in] pVoid connection
);
```

Qualifiers

- synchronous
- thread-safe

Description

Close the database connection

Parameters

Table 39

Name	Description
`connection`	Database connection to be closed.

Return Values

Default

Precondition

`connection = pointer to open database connection`

Action

None

Postcondition

Database `connection` is closed and resources are freed.

6.3.1.5.1.3 query

Signature

```
rcResult query (
  [in] pVoid connection,
  [in] String sql,
  [out] pVoid queryResult
);
```

Qualifiers

— synchronous

— thread-safe

Description

Execute a SQL[14] query.

Parameters

Table 40

Name	Description
`connection`	Database connection used for the query
`sql`	String expressing the query in SQL [14]
`queryResult`	Pointer to the result of the query

Return Values

Default

Precondition

```
connection = pointer of open database connection
```

```
sql = S
```

Action

None

Postcondition

```
queryResult = "result of executing query S"
```

6.3.1.5.1.4 fetchRow

Signature

```
rcResult fetchRow (
  [in] pVoid queryResult,
  [out] pVoid row
);
```

Qualifiers

— synchronous

— thread-safe

Description

Fetch the next row of the queryResult.

Parameters

Table 41

Name	**Description**
`queryResult`	Result of a previously executed query
`row`	Output parameter used to return the row

Return Values

Default

Precondition

```
queryResult = QR

pos ("position in the queryResult") = i

i < # rows of QR
```

Action

None

Postcondition

```
row = QR[i]

pos = i+1
```

6.3.1.5.1.5 nrOfRows

Signature

```
rcResult nrOfRows (
   [in] pVoid queryResult,
   [out] Int32 retValue
);
```

Qualifiers

— synchronous

— thread-safe

Description

Return the number of rows that are in the result of a Query.

Parameters

Table 42

Name	Description
`queryResult`	Result of a previously executed query
`retValue`	Output parameter used to return the number of rows

Return Values

Default

Precondition

```
queryResult = QR
```

Action

None

Postcondition

```
retValue = number of rows QR
```

6.3.1.5.1.6 fetchField

Signature

```
rcResult fetchField (
   [in] pVoid row,
   [in] String fieldId,
   [out] String *retValue
);
```

Qualifiers

— synchronous

— thread-safe

Description

Return a field from a row (of a query result)

Parameters

Table 43

Name	**Description**
`row`	Pointer to row (part of a queryResult)
`fieldId`	String that identifies the field. This can be through name or by number (starting by 0).
`retValue`	Output parameter used to return the value of the field

Return Values

Default

Precondition

`row = R`

`fieldId = id`

Action

None

Postcondition

`retValue = R[id]`

6.3.1.5.1.7 nrOfFields

Signature

```
rcResult nrOfFields (
  [in] pVoid row,
  [out] Int32 *retValue
);
```

Qualifiers

— synchronous

— thread-safe

Description

Return the number of fields of a row.

Parameters

Table 44

Name	**Description**
`row`	Row of a previously executed query
`retValue`	Output parameter used to return the number of fields of the row.

Return Values

Default

Precondition

`Row = R`

Action

None

Postcondition

`retValue = number of fields of R`

7 Realization overview

7.1 Introduction

This clause is informative. There is a need for integrity management mechanisms. Software systems are required to be robust and reliable. Currently software systems are continuously evolving; both during their development as well as during deployment. The M3W architecture provides means for the dynamic replacement, removal and addition of components in a deployed system, to cater for the increasing demand for flexibility. This facility endangers the software integrity of the system.

The integrity management mechanisms specified in this clause provides means for maintaining and restoring integrity in order to achieve robust and reliable device operation.

The realization is based on the core realization technology specified in ISO/IEC 23004-3.

7.2 System Integrity management mechanisms

System Integrity Management (SIM) can be divided into several activities:

— Monitoring: Retrieving the model describing the current configuration of a terminal (collecting data).

— Diagnosis: Based on the collected data by the monitoring activity faults in the configuration of the terminal are identified.

— Repairing: Faults identified by the diagnosis activity need to be removed. A repair plan for the removal of the fault needs to be generated and executed.

— *Prevention:* Prevention is an activity executed in parallel with the other three activities. During normal operation of a terminal, the software configuration can change. Components may be removed, added or replaced and applications can be downloaded or removed. Prevention is the set of checks that prevents the introduction of faults as part of the normal operation process.

Monitoring, Diagnosis and Repairing are closely related. The example scenario in Figure 6 illustrates this relation. System Integrity Management mechanisms can also remove known faults that have not been activated.

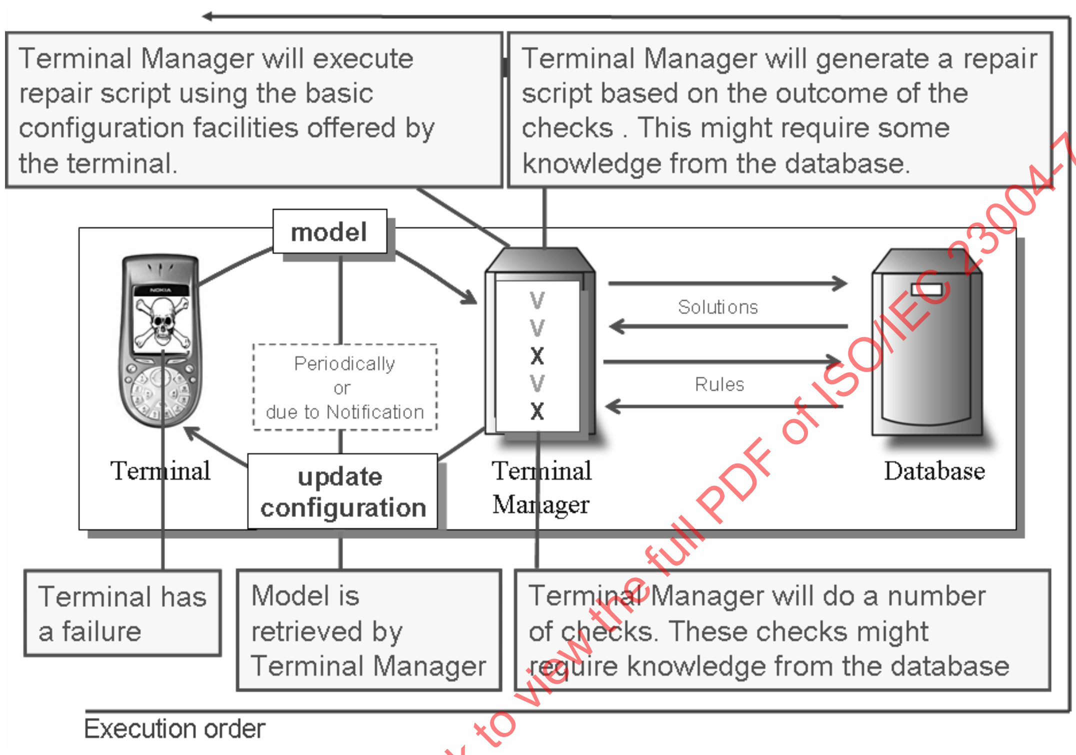
Terminal Manager will execute repair script using the basic configuration facilities offered by the terminal.
Terminal Manager will generate a repair script based on the outcome of the checks . This might require some knowledge from the database.
model
Periodically or due to Notification
update configuration
V V X V X
Solutions
Rules
Terminal
Terminal Manager
Database
Terminal has a failure
Model is retrieved by Terminal Manager
Terminal Manager will do a number of checks. These checks might require knowledge from the database
Execution order

Figure 6 — Example Integrity Management Scenario

System Integrity Management can serve as an escalation point for a Fault Management framework specified in ISO/IEC 23004-6.

7.3 Roles in System Integrity Management

Figure 6 illustrates that there are 3 roles involved in system integrity management:

— Terminal role: The main responsibilities of the Terminal role are to expose a model of the current software configuration and also to offer basic software configuration facilities.

— Terminal Manager role: The main responsibilities of the Terminal Manager role are to monitor (model retrieving), diagnosis (finding faults in the software configuration) and repair (generate and execute a repair script that consists of basic configuration actions).

— Database role: The main responsibility of the Database role is to provide knowledge which is used for diagnosis (rules) and also for repair (solutions).

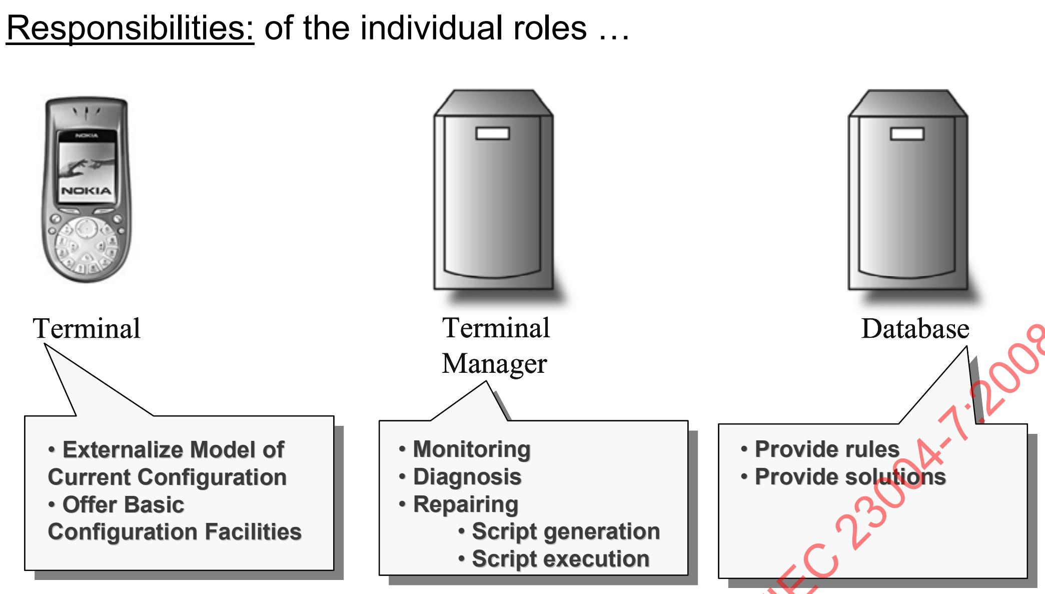

Figure 7 — Responsibilities of System Integrity Management roles

The terminal role will run on a terminal. The Terminal Manager role can run on the same terminal (Local System Integrity Management), or a different device (Remote System Integrity Management). The database role can be local or remote for the terminal manager. The database provides the rules for diagnosis and the solutions for the repair plan.

8 System Integrity Management realization

8.1 Structural view

We distinguish different roles, with main responsibilities.

— Terminal

  — Externalize self-model

  — Basic configuration operations of the terminal

— Terminal Manager

  — Retrieving model (elements)

  — Retrieving rules

  — Diagnosis

  — Generation of repair plan

— Database

  — Providing rules

  — Providing default solutions for problems

- Notification
  - Notification that there are problems, which can be used by, for example, the user and also the Fault Management framework.
    - Initiates monitoring, diagnosis and repair.

8.1.1 Terminal

8.1.1.1 Responsibilities

The terminal role is responsible for providing the model of the terminal to the outside world. These models can be used for monitoring and diagnosis. Furthermore the terminal provides mechanisms for execution of a number of actions. This can be used to execute for example a repair plan.

8.1.1.2 Provided Interfaces

The terminal role implements the `rcIModel` interface and the `rcITerminalSource` interface. The `rcIModel` interface can be used to inspect, update, and take elements from the self-model of the terminal. This interface consists of 3 operations:

```
interface rcIModel {CE752B0D-3795-D811-87C6-0008744C31AC} {
    void inspect(in String elementId, out String element);
    void update(in String elementId, in String value)
       raises RC_ERR_TERMINAL_NOT_ALLOWED = 0x00000001;
    void take(in String elementId, out String element);
       raises RC_ERR_TERMINAL_NOT_ALLOWED = 0x00000001;
  };
```

The operation `inspect()` has two parameters. An input parameter `elementId` that identifies the element(s) of the self model of the terminal that needs to be inspected. The self model of the terminal is a hierarchical XML model (see Annex D). The XPath language [13] is used to address parts of the self-model. The output parameter element is used to return the element(s) of the self-model that are inspected. The representation is an XML fragment.

The operation `update()` is used to update the elements part of the self-model that are not generated automatically based on the current configuration. This operation has two parameters. An input parameter `elementId` identifies the element(s) of the self model that are updated. The XPath language [13] is used for addressing parts of the self-model. The second input parameter is the `element` (value) that is being added. Updating elements part of the self-model that are automatically generated has no effect since, these are re-generated before every inspection.

The operation `take()` has the same parameters as the inspect operation. The `take` operation also returns the element identified by the `elementId` parameter similar to the `inspect()` operation, and subsequently removes the element identified by `elementId` from the self-model. Taking an automatically generated element of the self model comes down to inspecting it, since it will be added again before the next inspection.

The `rcITerminalSource` interface is also implemented by the terminal role. This interface is used to provide some basic operations to modify the configuration of the terminal. The `rcITerminalSource` consists of 10 operations:

```
interface rcITerminalSource {08D0F381-2099-D811-87C6-0008744C31AC} {
  void registerComponent(in pUUID cmpId, in String location)
     raises RC_ERR_TERMINAL_INVALID_LOCATION = 0x00000002,
     raises RC_ERR_TERMINAL_INVALID_COMPONENT = 0x00000004,
     raises RC_ERR_TERMINAL_ALREADY_REGISTERED = 0x00000008;
  void unregisterComponent(in pUUID cmpId);
  void registerService(in pUUID cmpId, in pUUID svcId)
     raises RC_ERR_TERMINAL_ALREADY_REGISTERED = 0x00000008,
     raises RC_ERR_TERMINAL_UNKNOWN_COMPONENT = 0x00000010;
  void unregisterService(in pUUID svcId);
  void registerComplies(in pUUID complying, in pUUID blueprint)
     raises RC_ERR_TERMINAL_NOT_ALLOWED = 0x00000001;
  void unregisterComplies(in pUUID complying, in PUUID blueprint);
  void installModel(in pUUID m3wCmpId, in pUUID mdlId, out String location)
     raises RC_ERR_TERMINAL_INVALID_LOCATION = 0x00000002,
     raises RC_ERR_TERMINAL_NOT_ALLOWED = 0x00000001;
  void uninstallModel(in pUUID mdlId)
     raises RC_ERR_TERMINAL_NOT_ALLOWED = 0x00000001;
  void executeModel(in pUUID mdlId)
     raises RC_ERR_TERMINAL_NOT_ALLOWED = 0x00000001;
  void downloadM3WComponent(in pUUID m3wCmpId);
  void removeM3WComponent(in pUUID m3wCmpId);
};
```

The `registerComponent()` operation can be used to register new executable components with the M3W runtime environment. This operation has two input parameters. The first parameter is the `uuid` of the executable component to be registered and the second is the `location` of the executable component.

The `unregisterComponent()` operation de-registers an executable component from the M3W runtime environment. This operation takes one input parameter `cmpId`.

The `registerService()` operation 'adds' a service to an executable component in the M3W registry. This operation takes two input parameters. The first parameter (`cmpId`) identifies the executable component to which the service is added. The second parameter is the uuid of the service that is added (`svcId`).

The `unregisterService()` operation un-registers a service from the M3W runtime environment. This operation takes one input parameter `svcId` identifying the service that will be unregistered.

The `registerComplies()` operation is used to register new complies relations with the M3W registry. This operation takes two input parameters. The first parameter (`complying`) identifies the compliant entity. The second parameter (`blueprint`) identifies the template entity.

The `unregisterComplies()` operation removes complies relations from the M3W registry. This operation takes the same parameters as the `registerComplies` relation.

The `installModel()` operation is used to extract a model from a M3W component and install it on the terminal. This operation takes 2 input parameters and 1 output parameter. The first input parameter (`m3wCmpId`) identifies the M3W component which contains the model that needs to be installed. The second input parameter (`mdlId`) identifies the model that needs to be installed. The output parameter (`location`) holds the location where the model is installed.

The `uninstallModel()` operation is used to uninstall a model from a terminal. This operation takes one input parameter (`mdlId`), the uuid of the model that needs to be uninstalled.

The `executeModel()` operation is used to execute models. This is not always possible since not all models are executable. Typical examples of models that are executable and for which this operation is useful are:

— registration scripts

— configuration scripts

— ...

The `executeModel()` takes one input parameter (`mdlId`). This is the uuid of the model that needs to be executed.

The `downloadM3WComponent()` operation is used to download a M3W component to a terminal. Calling this operation will result in initiating the download using the required `IInitiator` interface. This operation has one parameter (`m3wCmpId`), the uuid of the M3W component that needs to be downloaded.

The `removeM3WComponent()` operation is used to remove M3W components from the terminal. This operation takes one input parameter (`m3wCmpId`), the uuid of the M3W component that needs to be removed. All installed models of this M3W component remain on the terminal.

8.1.1.3 Implementation of the interfaces

```
service STerminal {DA459115-3895-D811-87C6-0008744C31AC} {
    provides {
        rcIModel model;
        rcITerminalSource source;
    };
    requires {
        rcITarget target;
        rcIInitiator initiator;
    };
};
```

8.1.2 Terminal Manager

8.1.2.1 Responsibilities

The terminal manager role is responsible for monitoring, diagnosing and repairing a terminal. Monitoring is quite straight forward as this involves retrieving the model of the terminal that is made external by the terminal role.

8.1.2.2 Provided Interfaces

The Terminal Manager Role implements the `rcITerminalManager` interface. This interface consists of 3 operations:

```
interface rcITerminalManager {04ABEB2C-3795-D811-87C6-0008744C31AC} {
    void monitor(out String model, out Bool possibleProblem);
    void diagnose(in String model, out String diagnosis);
    void repair(in String model, in String diagnosis,
                out Int32 remainingFaults);
};
```

The operation `monitor()` is used to detect whether there is a problem (`possibleProblem`). It can be considered as a quick first check. This operation has 2 output parameters. The first parameter `model` is a string that represents the self-model that is created during the monitoring. The second output parameter `possibleProblem` indicates whether there possibly is a problem with the terminal.

The operation `diagnosis()` is used to detect what is wrong with the terminal. During execution of this operation, the self-model is examined and a diagnosis report is generated. This operation has two parameters, one input parameter `model` which contains the self-model of the terminal. The second is an output parameter `diagnosis` which is used to return the diagnosis report generated during the operation. The diagnosis report is an xml fragment (see the schema definition in Annex D).

The operation `repair()` is used to fix the faults that are detected during diagnosis (and are in the diagnosis report). The repair operation has two input parameters and one output parameter. The self-model (`model`) and the diagnosis report (`diagnosis`) are input for the repairing. The output parameter `remainingFaults` is used to return whether the repairing was successful or not. `remainingFaults` returns the number of faults identified by the diagnosis report that are not removed.

8.1.2.3 Implementation of interfaces

```
service STerminalManager {B4CD1223-3895-D811-87C6-0008744C31AC} {
    provides {
        rcITerminalManager terminalManager;
    };
    requires {
        rcIModel model;
        rcITerminalSource source;
        rcIDatabase rules;
    };
};
```

8.1.3 Database

8.1.3.1 Responsibilities

The database is responsible for storing and providing information necessary for diagnosis and repair plan generation.

8.1.3.2 Provided Interfaces

The database role implements the `rcIDatabase` interface. This interface wraps the basis functionality of a database client.

```
interface rcIDatabase {E421F248-1E99-D811-87C6-000EA69DB1A4} {
    void open(out pVoid connection);
    void close(in pVoid connection);
    void query(in pVoid connection, in pVoid sql, out pVoid queryResult);
    void fetchRow(in pVoid queryResult, out pVoid row);
    long nrOfRows(in pVoid queryResult);
    string fetchField(in pVoid row, in String fieldId);
    long nrOfFields(in pVoid row);
};
```

8.1.3.3 Implementation of interfaces

```
service SDatabase {F2F215F8-2099-D811-87C6-000EA69DB1A4} {
    provides {
        rcIDatabase rules;
    };
};
```

8.1.4 Notification

8.1.4.1 Responsibilities

The notification role is a feature which can be used to indicate that there is a need for integrity management on a device. This need can be expressed by the user of the device as well as by the Fault Management framework.

8.1.4.2 Provided Interfaces

The notification role implements the `rcINotify` interface. This interface provides a trigger which initiates monitoring, diagnosis and repair by the System Integrity Management framework.

```
interface rcINotify {3eba62df-8ab8-4832-be31-40f871874eb3} {
    void notify();
    void notifyWithServiceId(in pUUID svcId);
};
```

The operation `notify()` is used to initiate monitoring, diagnosis and repair. This operation does not take any input parameters. Monitoring, diagnosis and repair is executed on a separate thread of control. Therefore the client is not blocked.

The operation `notifyWithServiceId()` is also used to initiate monitoring, diagnosis and repair but this operation also takes a service identifier (`svcId`). This parameter contains the service in which an error occurred. This information aids in the diagnosis process. This operation does not block the client during monitoring, diagnosis and repair because these activities are executed on a different thread of control.

8.1.4.3 Implementation of interfaces

```
service SNotify {b2bb9f99-3dfa-4282-bbe0-5ad59bf7a622} {
    provides {
        rcINotify notify;
    };
    requires {
        rcITerminalManager terminalManager;
    };
};
```

8.2 Behavior View

8.2.1 Monitoring

Monitoring involves interaction between a terminal and the terminal manager. Through monitoring we aim to make the internal state and context of the terminal externally visible. The state and context of a terminal is transferred using a model.

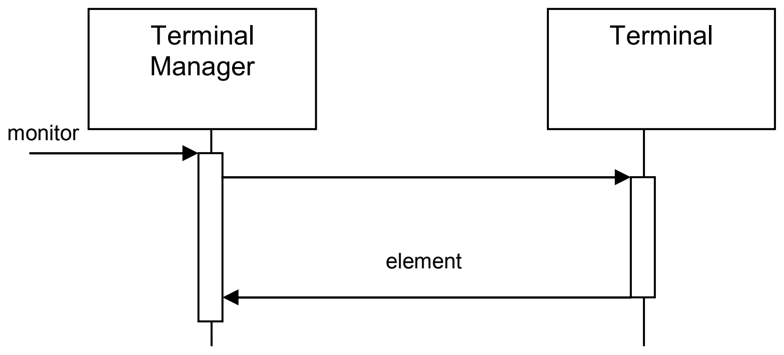

Figure 8 — Monitoring Message Sequence Chart

8.2.1.1 Model of the terminal

In this subclause we describe what information is part of the model of a terminal. This information can be extended. This model of the terminal contains a description of the software and hardware of the terminal. Therefore it contains a description of the:

— Application layer

— Middleware layer

— Platform layer

The models of for the terminal will be transferred using XML.

8.2.1.1.1 Application layer

The application layer model contains the following information:

```
application layer    =   A
                         where A is a set of a (Applications)
a                    =   <name,version,D>
                         where D is a set of Services (dependencies)
```

8.2.1.1.2 Middleware layer

The model for the middleware layer contains the following information:

```
middleware layer =   <runtime,R,C>
                     where R is set of r (registered Components)
                     and C is set of c (Complies Relations)
runtime          =   version
r                =   <P>
                     where P is set of p (contained Services)
p                =   Service
c                =   <compliant,template>
                     where compliant and template are Services or Libraries
```

8.2.1.1.3 Platform layer

The model for the platform layer contains the following information:

```
platform  layer  =  <os,cpu,storage>
os               =  <name,version>
cpu              =  <vendor,family,model,clockspeed,cache size>
storage          =  <memory size,swap size,F>
                    where F is set of f (filesystems)
f                =  <name,size of filesystem>
```

8.2.2 Diagnosis

Diagnosis involves doing checks based on the state and context externalized by a terminal and models of a M3W component. Based on the type of checks that are executed the terminal manager might need information from a database. This information can be a blacklist of services, a preferred structure, etc.

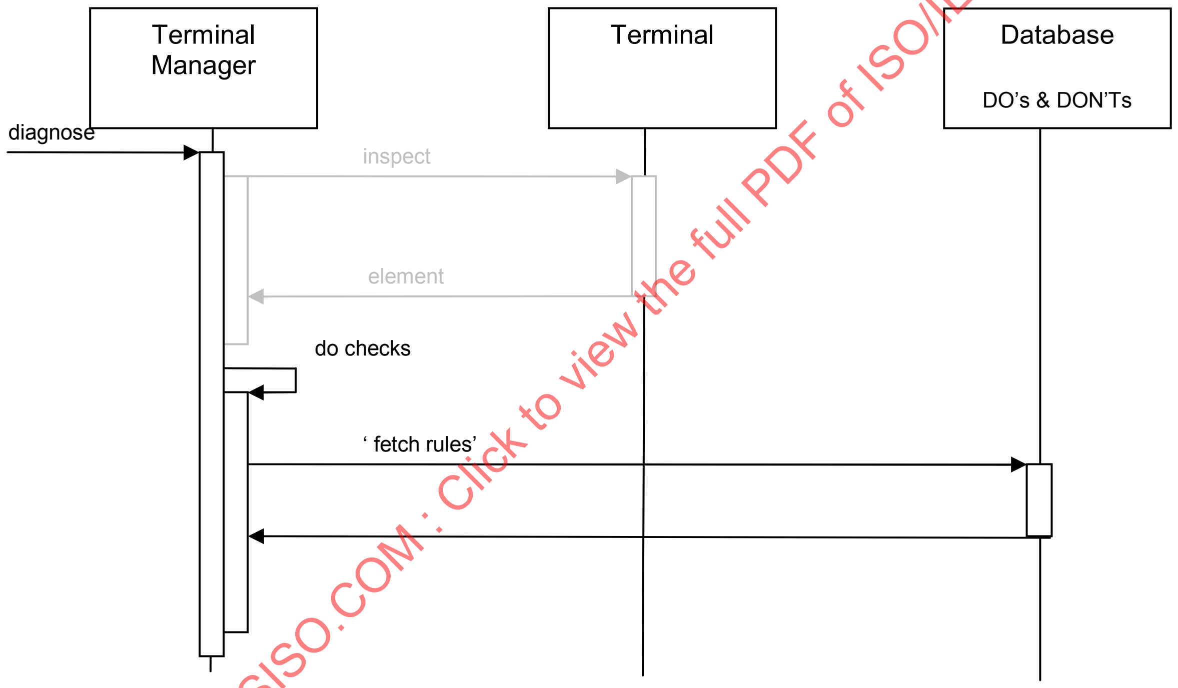

Figure 9 — Diagnosis Message Sequence Chart

In Annex B we present a number of example checks.

8.2.3 Repairing

Faults that are identified during the diagnosis can sometimes be repaired by the terminal manager. A repair plan is generated for example depending on the type of fault and a number of known solutions. After generation of the repair plan, this plan is executed therefore the terminal manager uses the basic configuration functionality provided by the terminal.

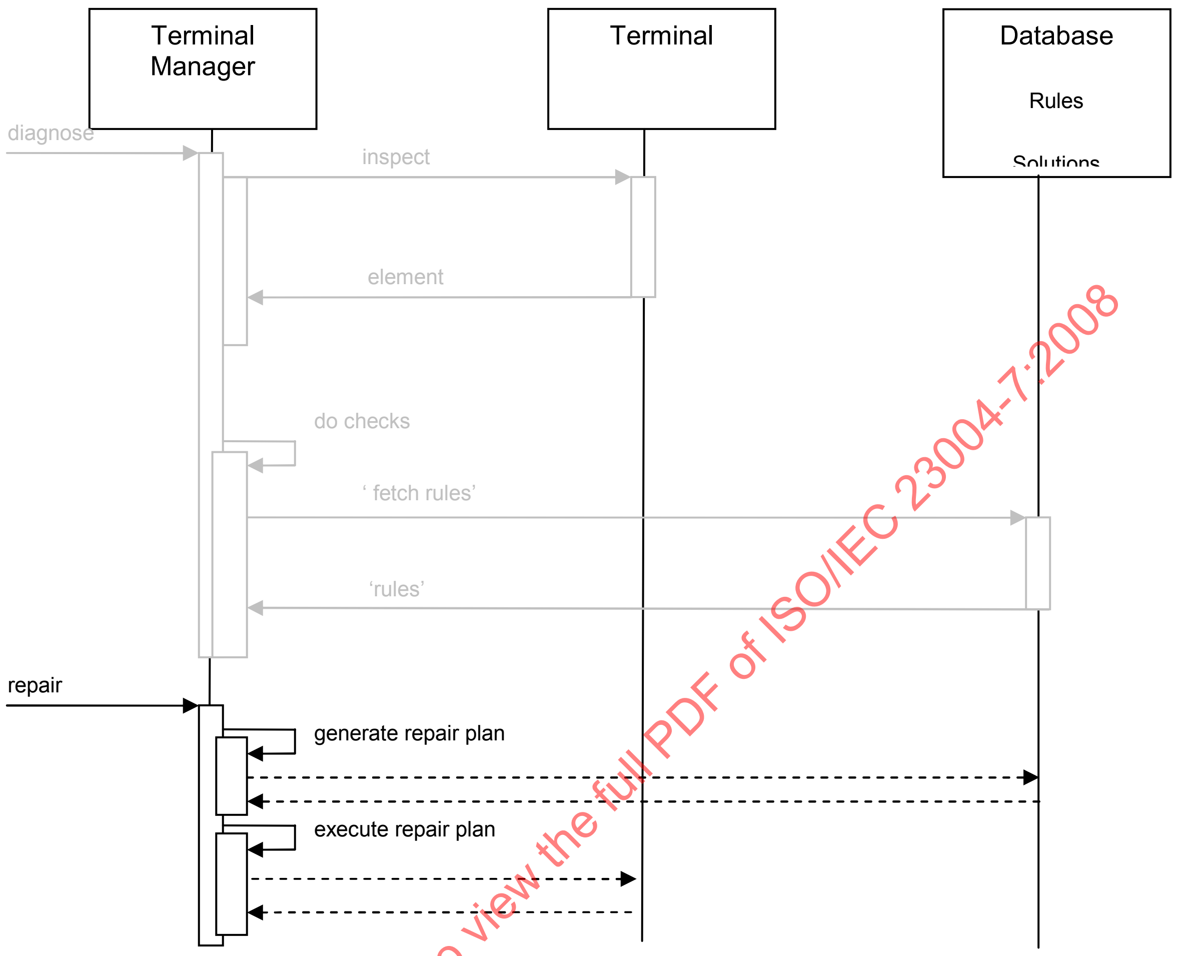

Figure 10 — Repairing Message Sequence Chart

8.2.4 Notification

After notification by, for example, the user or the Fault Management framework, control will be returned to the invoking party and a separate thread started for monitoring (see 8.2.1), diagnosis (see 8.2.2) and repair (see 8.2.3).

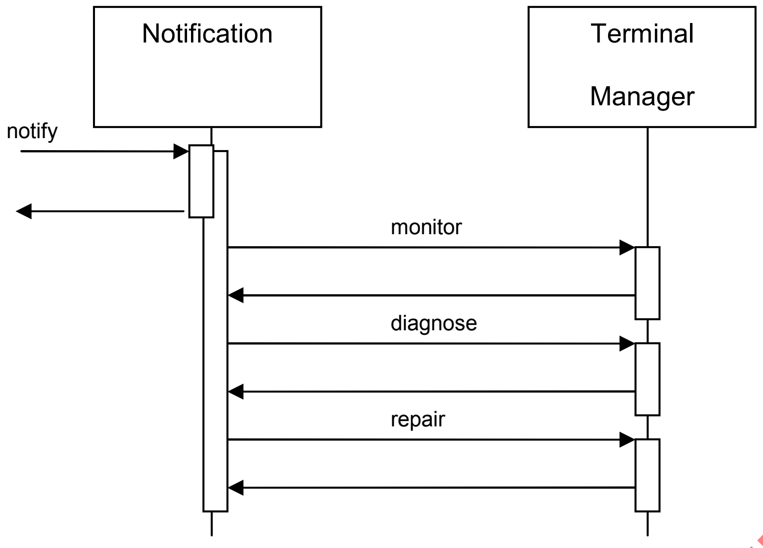

Figure 11 — Notification Message Sequence Chart

8.2.5 Prevention

Prevention will try to prevent the introduction of faults during normal operation. As part of the M3W a download framework has been specified (ISO/IEC 23004-5). This download framework is responsible for downloading suitable components to a terminal when this is required. Part of the download framework is the decider role. The decider is responsible for the decision whether a component is suitable for a terminal.

The decider uses (a selection of) the checks that are also used for diagnosis, to assess suitability of a component for a terminal. Since these checks are based on knowledge that evolves over time, it is useful to also do these checks during diagnosis.

8.3 Deployment view

The roles discussed in subclause 8.1 can be deployed on different devices. The terminal role must be deployed on the terminal. The terminal manager role and the database can be deployed on different devices.

System integrity management can be done locally on a terminal. In that case all the roles are deployed on the terminal.

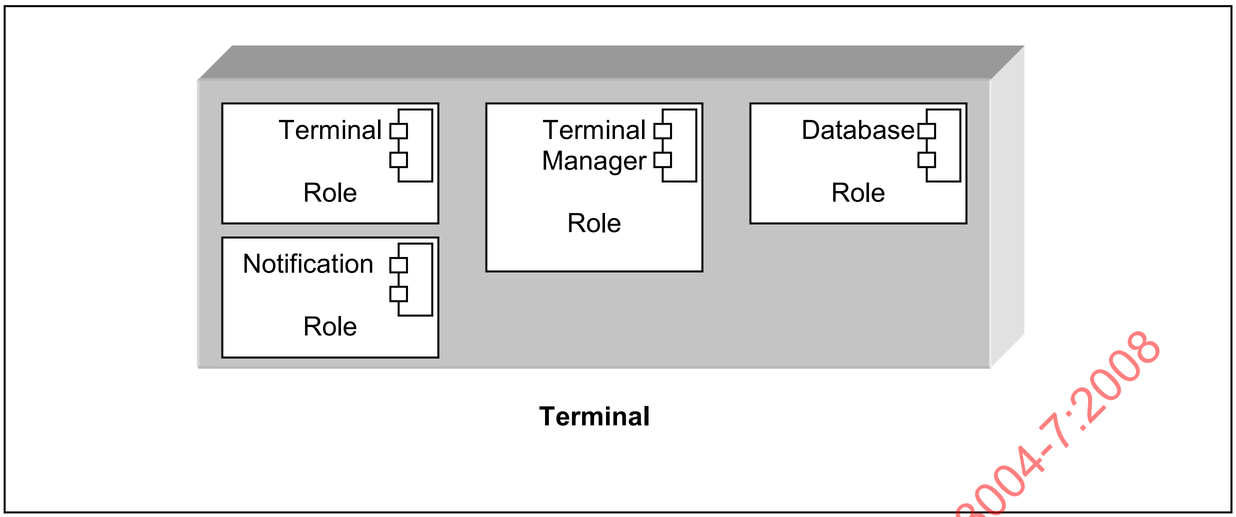

Figure 12 — Deployment - Terminal

System integrity management can also be performed from a remote device (not the terminal). In this case, the terminal manager role and the database are not deployed on the terminal, but on (a) different device(s).

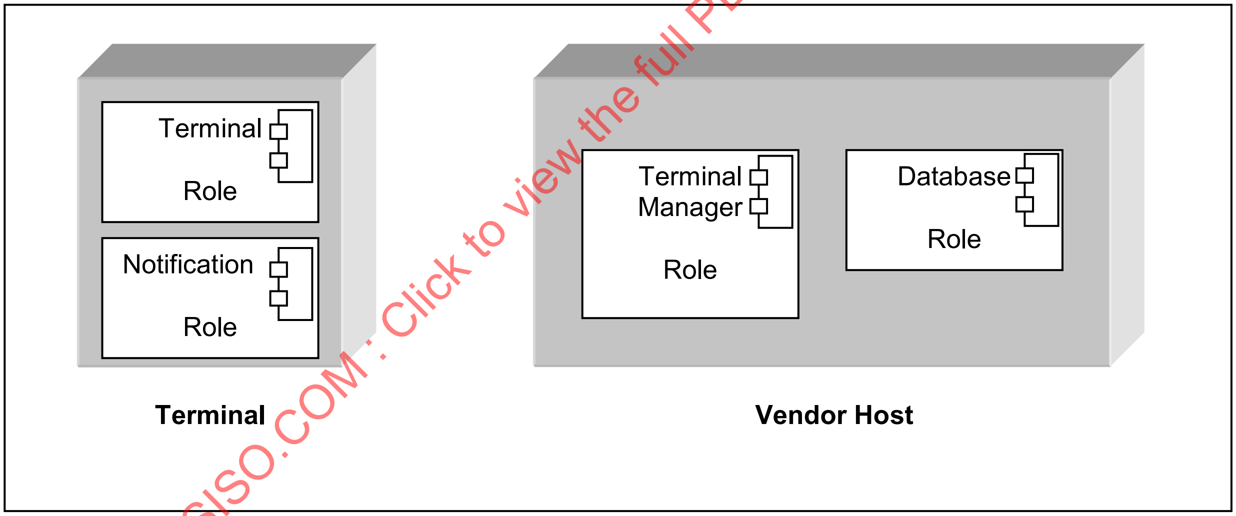

Figure 13 — Deployment - Vendor Host