
**Information technology — Multimedia
Middleware —**

**Part 1:
Architecture**

*Technologies de l'information — Intergiciel multimédia —
Partie 1: Architecture*



Reference number
ISO/IEC 23004-1:2007(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2007

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2007

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword	iv
Introduction.....	v
1 Scope	1
2 Organization of this document	1
3 Normative references	2
4 Terms and definitions	2
4.1 Specification terms and definitions.....	2
4.2 Realization terms and definitions	7
5 M3W architecture.....	8
5.1 General	8
5.2 Context	8
5.3 API specification.....	9
5.4 Realization technology	10
5.5 Realization.....	16
Annex A (informative) API specifications reader's guide	18
Annex B (normative) Basic types and constants	37
Annex C (normative) API evolution rules	43
Annex D (informative) Naming conventions	58
Annex E (informative) Constraints on execution architecture	64
Annex F (informative) Error handling	69
Annex G (informative) Notification	76
Annex H (informative) Get set patterns	99
Annex I (informative) Handling variation	121
Annex J (informative) API qualifiers	129
Annex K (informative) Name abbreviations guide	131
Annex L (informative) IDL.....	135
Bibliography.....	146

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 document 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 23004-1 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*
- *Part 3: Component model*
- *Part 4: Resource and quality management*
- *Part 5: Component download*
- *Part 6: Fault management*
- *Part 7: System integrity management*

Introduction

MPEG, ISO/IEC JTC 1/SC 29/WG 11, has produced many important standards (MPEG-1, MPEG-2, MPEG-4, MPEG-7, and MPEG-21). MPEG feels that it is important to standardize an application programming interface (API) for Multimedia Middleware (M3W) that complies with the requirements found in the annex to the Multimedia Middleware (M3W) Requirements Document Version 2.0 (ISO/IEC JTC1/SC 29/WG 11 N 6981).

The objectives of MPEG Multimedia Middleware (M3W) are to allow application software to execute multimedia functions with a minimum knowledge of the inner workings of the multimedia middleware, and to allow the triggering of updates to the multimedia middleware to extend the API. The first goal can be achieved by standardizing the API that the multimedia middleware offers. The second goal is much more challenging, as it requires mechanisms to manage the multimedia middleware components, and to ensure that these updates can be integrated in a controlled and dependable manner.

This part of ISO/IEC 23004 provides the following:

- a *vision* for a multimedia middleware API framework that enables
 - application software to control and extend multimedia middleware in a standardized manner;
 - multimedia software to be easily developed for, and deployed across, a variety of platforms;
 - the transparent and augmented use of multimedia resources across a wide range of networks and devices, to optimize the perceived quality for users;
- a method to facilitate the integration of APIs to software components and services in order to harmonize *technologies* for the creation, management, manipulation, transport, distribution and consumption of content;
- a *strategy* for achieving a multimedia API framework by the development of specifications and standards based on well-defined functional requirements through collaboration with other bodies.

Information technology — Multimedia Middleware —

Part 1: Architecture

1 Scope

This part of ISO/IEC 23004 defines the architecture of the MPEG Multimedia Middleware (M3W) technology.

2 Organization of this document

The remainder of this part of ISO/IEC 23004 is structured as follows. Clause 3 gives an overview of the references that are indispensable for the application of this part of ISO/IEC 23004. Clause 4 gives an overview of the terms and definitions used in this part of ISO/IEC 23004.

Clause 5 describes the high level architecture of a complete M3W system. The M3W middleware is part of an M3W system, and ISO/IEC 23004-2 specifies the application programming interface (API) of M3W as well as the realization technology. Subclause 5.2 contains a description of the context of M3W (an M3W system). This subclause also introduces the distinction between M3W API specification and realization of M3W. Subclause 5.3 gives an overview of the M3W API specification. Subclause 5.4 gives an overview of the M3W realization technology that is specified in ISO/IEC 23004-3, ISO/IEC 23004-4, ISO/IEC 23004-5, ISO/IEC 23004-6 and ISO/IEC 23004-7. Subclause 5.5 briefly discusses realization of the M3W, and emphasizes that developers can differentiate their software by producing different realizations.

This part of ISO/IEC 23004 has the following annexes.

Annex A, API specifications reader's guide, explains how the functional and the support parts of the API are specified.

Annex B, Basic types and constants, gives an overview of the basic types and constants that are used in the API specification and the realization technology.

Annex C, API evolution rules, lists the rules for the evolution of API specifications.

Annex D, Naming conventions, lists the naming conventions used in the API specifications and the realization technologies.

Annex E, Constraints on execution architecture. In the API specification a number of assumptions are made on the execution architecture. This annex lists the assumptions which hold, unless specified otherwise.

Annex F, Error handling, describes the default error handling mechanism in M3W.

Annex G, Notification, describes the default notification mechanism in M3W.

Annex H, Get set patterns, describes the default way of dealing with 'get set' patterns in M3W.

Annex I, Handling variation, explains how to deal with variation in M3W systems.

Annex J, API qualifiers, contains a table that lists all of the qualifiers that are used in the API specification.

Annex K, Name abbreviations guide, gives an alphabetical list of words and their abbreviations commonly used in names.

Annex L, IDL, describes the two variations of IDL used in ISO/IEC 23004. One variation is used for the specification of the M3W API, the other one is used in the realization technology. This annex explains how the IDL used to specify the M3W API can be translated into the IDL used in the realization technology.

3 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 23004-2, *Information technology — Multimedia Middleware — Part 2: Multimedia API*

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

ISO/IEC 23004-4, *Information technology — Multimedia Middleware — Part 4: Resource and quality management*

ISO/IEC 23004-5, *Information technology — Multimedia Middleware — Part 5: Component download* ¹⁾

ISO/IEC 23004-6, *Information technology — Multimedia Middleware — Part 6: Fault management* ¹⁾

ISO/IEC 23004-7, *Information technology — Multimedia Middleware — Part 7: System integrity management* ¹⁾

4 Terms and definitions

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

4.1 Specification terms and definitions

4.1.1

API specification

M3W API specification, which defines a collection of software interfaces providing access to coherent streaming-related functionality

4.1.2

interface suite

collection of mutually related interfaces providing access to coherent functionality

4.1.3

logical component

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

4.1.4

role

abstract class, i.e. a class without implementation defining behavior only

4.1.5

role instance

object playing a role, i.e. an object displaying the behavior defined by the role

1) To be published.

4.1.6**attribute**

instance variable associated with a role

NOTE Attributes are used to associate state information with roles.

4.1.7**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.

4.1.8**specification item**

entity defined in a specification

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

4.1.9**IDL**

Interface Definition Language

4.1.10**qualifier**

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

4.1.11**constraint**

restriction that applies to a specification item

4.1.12**execution constraint**

constraint on multi-threaded behavior

4.1.13**model type**

data type used for specification (modeling) purposes only

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

4.1.14**model constant**

constant used for specification (modeling) purposes only

4.1.15**enum element type**

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

4.1.16**enum set type**

32-bit integer data type representing sets of enumerated values

4.1.17**set type**

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

NOTE Unlike enum sets, these sets may be infinite.

4.1.18

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. Unlike arrays, the domain and range types may be arbitrary, and possibly infinite types.

4.1.19

entity type

class of objects that may have attributes associated with them

4.1.20

interface-role model

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

4.1.21

logical component instance

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

4.1.22

provides interface

interface that is provided by a role or role instance

4.1.23

requires interface

interface that is used by a role or role instance

4.1.24

specialization

behavioral 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.

4.1.25

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

4.1.26

mandatory interface

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

4.1.27

optional interface

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

4.1.28

configurable item

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

4.1.29

diversity attribute

role attribute that represents a configurable item

4.1.30**instantiation**

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

4.1.31**initial state**

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

4.1.32**observable behavior**

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

4.1.33**function behavior**

behavior of the functions in the provides interfaces of a role

4.1.34**streaming behavior**

input-output behavior of the streams associated with a role

4.1.35**active behavior**

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

4.1.36**instantiation behavior**

behavior of a role at instantiation time of a logical component

4.1.37**independent attribute**

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

4.1.38**dependent attribute**

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

4.1.39**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.

4.1.40**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.

4.1.41**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

4.1.42**event notification**

act of reporting the occurrence of events to 'interested' objects

4.1.43

event subscription

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

4.1.44

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.

4.1.45

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.

4.1.46

non-standard event notification

event notification that is accompanied by other actions (such as state changes of the notifying logical component)

4.1.47

client role

role modeling the users of a logical component

4.1.48

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

4.1.49

control interface

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

4.1.50

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

4.1.51

specialized interface

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

4.1.52

precondition

assertion that should be true immediately before a function is called

4.1.53

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.

4.1.54

out-call

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

4.1.55**postcondition**

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

4.1.56**asynchronous function**

function with a delayed effect

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

4.2 Realization terms and definitions**4.2.1****appliance**

product as seen by the customer, consisting of the device, an operating system (OS), components and applications

4.2.2**application**

software entity that provides a set of functions to a user

4.2.3**component**

unit of trading that conforms to the M3W component model

NOTE In the M3W component model, components are containers for models.

4.2.4**component model**

specification of what constitutes a component, defining *inter alia* the interaction mechanisms between components and between components and their environment

4.2.5**device**

physical part of the appliance, sometimes also used to denote an identifiable element of that appliance

4.2.6**executable component**

model that is executable on a platform

NOTE The executable component is a container for services.

4.2.7**M3W system**

system that conforms to the M3W specification

4.2.8**operational**

state when an M3W system is fulfilling its required functions for a period of time

4.2.9**platform**

operating system (OS) and hardware that executes the OS

4.2.10**system**

combination of a platform, a set of executable components, a run-time environment and a set of applications that can provide a user with a set of functions

NOTE A platform provides access to the underlying hardware, executable components contain services that provide advanced domain logic, and applications use the logic in the services to provide functions to a user.

5 M3W architecture

5.1 General

This clause gives an overview of the M3W software architecture. In subclause 5.2 we give an overview of the high level software structure of an M3W system, and the separation between the specification of the M3W API and its realization. Subclause 5.3 gives an overview of the specification of the M3W API. Subclause 5.4 gives an overview of the realization technology that shall be used to realize M3W.

5.2 Context

5.2.1 System overview

In an M3W system there are 3 distinct layers:

- Applications: The type of applications will depend on the appliance.
- Middleware layer: This layer consists of M3W and other middleware. M3W can be separated into two parts:
 - Functional part: This provides applications and other middleware with a multimedia platform.
 - Non-functional part: This provides a means to manage lifetime of, and interaction with, realization entities. Furthermore this enables management of non-functional properties, e.g. resource management, fault management and integrity management;
- Computing platform: The platform will depend on the appliance. The API of the computing platform is outside of the scope of M3W. Different realizations of M3W for different platforms will use the different API's for these platforms.

The layers mentioned above are not strict in the sense that applications and other middleware cannot use the computing platform directly. However, the latter is not recommended. The software structure of an M3W system is shown in Figure 1 — Software structure of an M3W system.

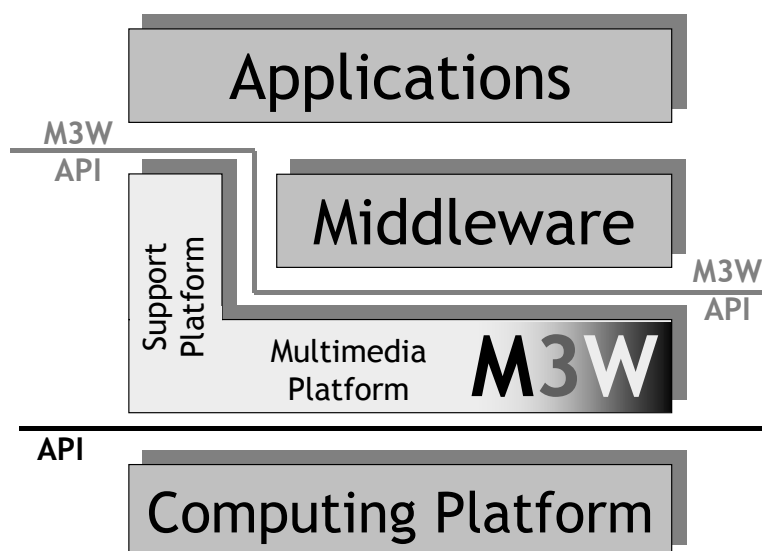


Figure 1 — Software structure of an M3W system

The scope of this standard is limited to the specification of the M3W API and the specification of the realization technology for M3W.

5.2.2 API specification versus realization

In subclause 5.2.1 we introduced the difference between the specification of the M3W API and the realization (technology) that provides an implementation for M3W. The specification of the M3W API provides an uniform view for the applications and other middleware. This uniform view enables the development of applications and other middleware independent of a specific realization of M3W.

The ability to create multiple different realizations for the M3W API enables vendors of M3W (or parts of M3W) to differentiate themselves from competitors.

In order to guarantee the interoperability of multiple parts of M3W provided by different parties, the same realization technology shall be used. Figure 2 — Specification and realization of M3W, illustrates the difference between:

- specification of the M3W API in terms of logical components and roles (see Annex A); and
- realization elements.

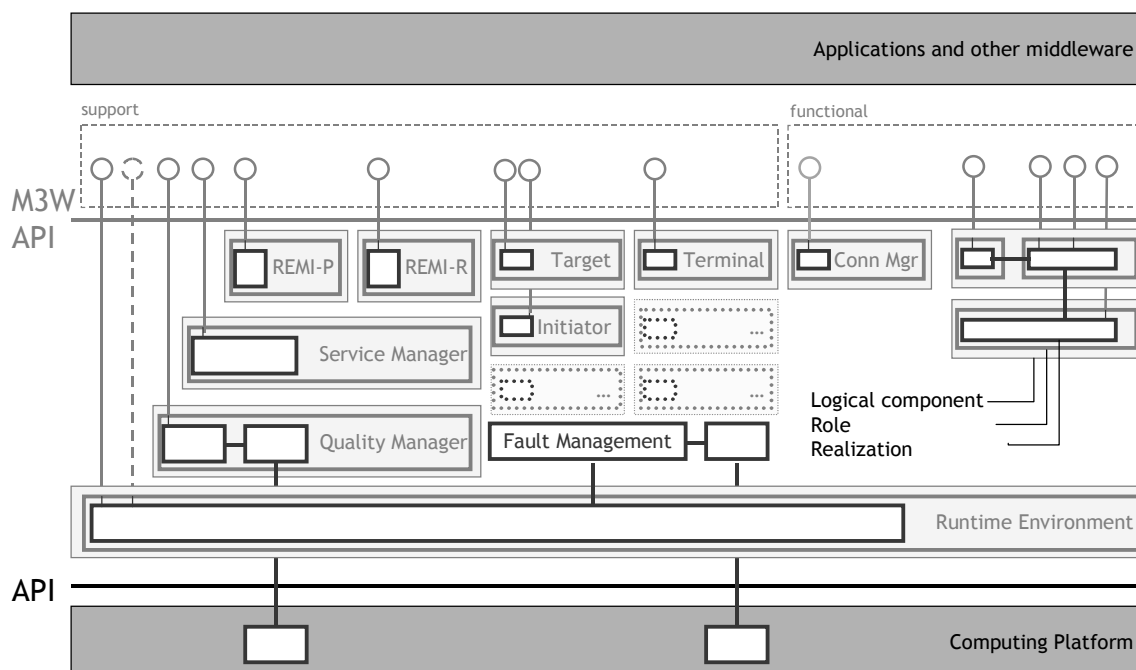


Figure 2 — Specification and realization of M3W

NOTE Figure 2 — Specification and realization of M3W, shows an example realization. Other realizations are possible as long as they comply with the M3W API specification, and the realization technology

5.3 API specification

The specification of the M3W API gives a uniform view to an M3W system. This view is described in terms of logical components and roles (see Annex A), and it should be noted that this view does not contain any realization elements. The M3W API specification consists of the specification of logical components and roles. These logical components and roles give a uniform view to the functionality independent of the realization of this functionality. These logical components and roles can be divided as follows:

- Support:
- Runtime Environment: enables creation of realization elements;

- Service Manager: enables creation of a realization for a logical component. This realization can consist of a number of connected realization elements;
- REMI-P and REMI R: gives a uniform approach to interaction between entities on different M3W systems;
- Quality Manager: enables the optimization of the overall Quality of Service (QoS) perceived by the user;
- Target, Initiator, Repository, Decider, and Locator: enable the transfer of new components to permanent storage of the M3W system;
- Terminal, Terminal Manger, Database: enable maintaining the consistency of the configuration of M3W;
- Functional: provide multimedia functionality to the applications and other middleware.

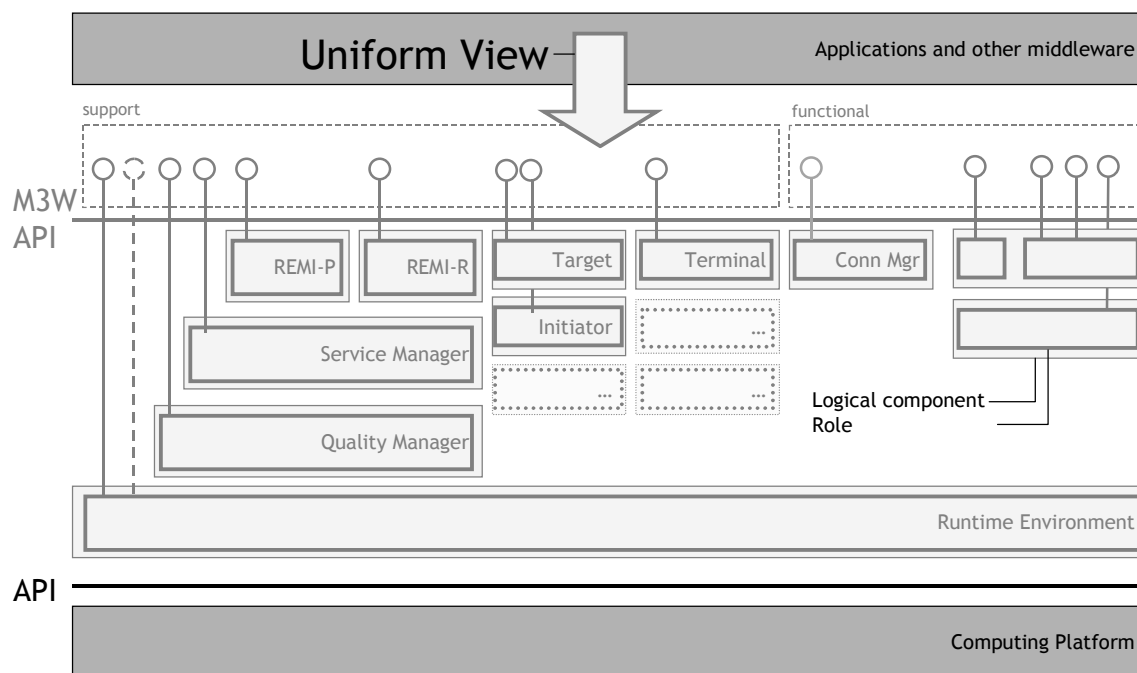


Figure 3 — Specification of the M3W API

The functional part of the M3W API mentioned above is specified in ISO/IEC 23004-2. The support part of the M3W API mentioned above, as well as the specification of the realization technology, are specified in ISO/IEC 23004-3, ISO/IEC 23004-4, ISO/IEC 23004-5, ISO/IEC 23004-6 and ISO/IEC 23004-7.

5.4 Realization technology

5.4.1 Overview

This subclause gives an overview of the realization technology that shall be used to realize M3W. The core of M3W is the component model and core framework, and this can be extended with a number of optional frameworks:

- Resource Management: manage the perceived Quality of Service;
- Download: enable transfer of new components to permanent storage of an M3W system;
- Integrity Management: maintain software consistency in M3W; and
- Fault Management: enable insertion of fault tolerance techniques.

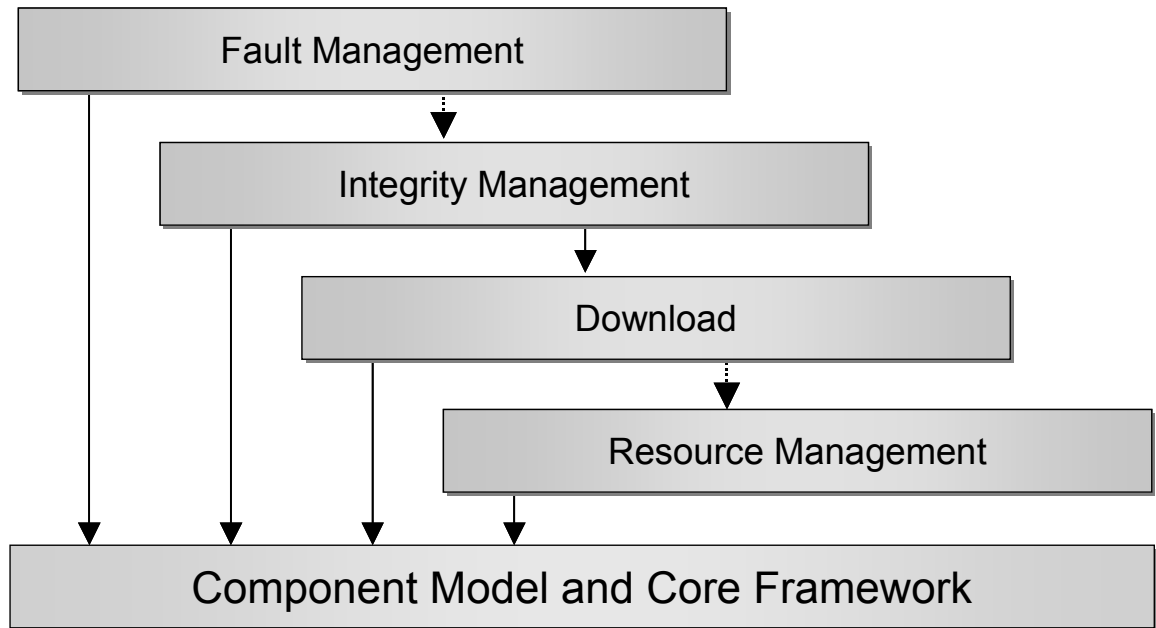


Figure 4 — M3W frameworks

All of these frameworks are discussed below.

5.4.2 Component Model and Core Framework

The core component model and core framework enable management of the life time of the realization elements and the interaction between these elements. The core framework provides a number of logical components:

- Runtime Environment;
- Service Manager; and
- REMI-P and REMI-R.

The Runtime Environment and Service Manager deal with the creation (instantiation) of realization elements. The Runtime Environment enables the creation of a single realization element (instantiation of services as is described in ISO/IEC 23004-3). The Service Manager enables the creation of (a number of connected) realization elements that realize a logical component.

The component model specifies what constitutes a component, executable component and service (see ISO/IEC 23004-3). Furthermore it specifies the interaction mechanisms for these entities, this includes:

- Binding
- Access to attributes of realization elements
- Operation invocation
- Navigation between interfaces

In short it specifies how to create realization elements and interact with them.

REMI-P deals with giving remote access to interfaces of realization elements (services) on an M3W system to realization elements on another M3W system. REMI-R deals with getting access to interfaces of realization elements on another M3W system.

Figure 5 — M3W core framework, depicts the main elements of the core framework (specification as well as realization). The core component model and core framework is specified in ISO/IEC 23004-3.

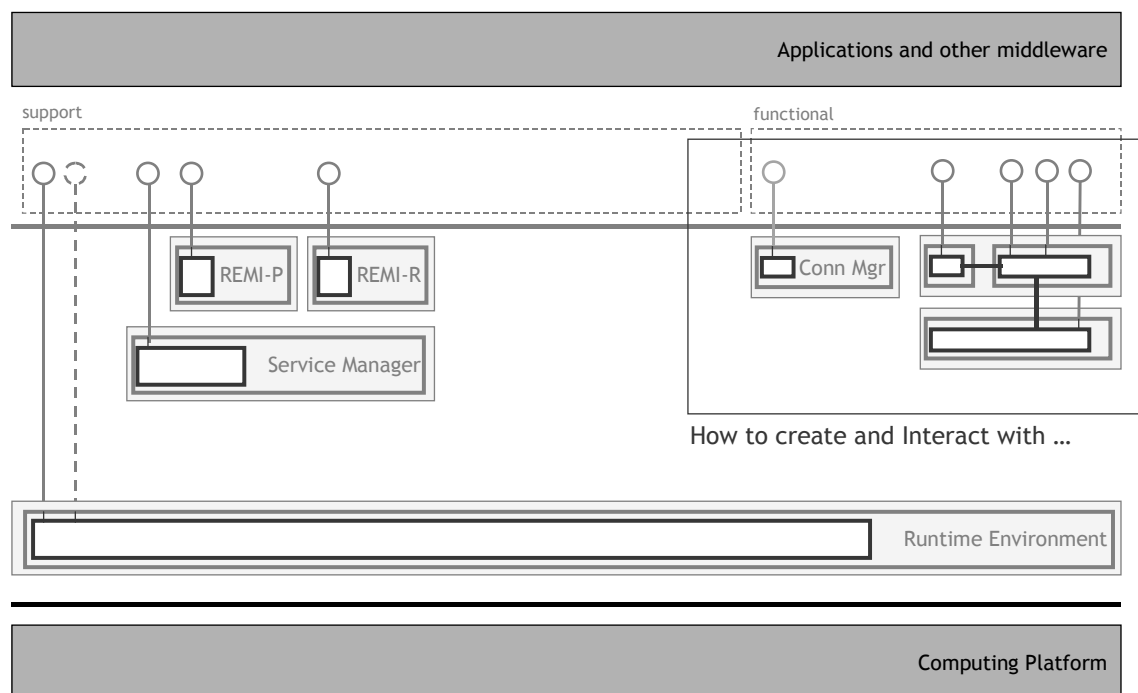


Figure 5 — M3W core framework

5.4.3 Resource Management

An optional addition to the core framework is the Resource Management Framework. This framework enables the optimization of the Quality of Service for the user of the appliance. In practice this will usually mean the optimization of the Quality of Service delivered by the functional part.

In order to be able to manage the Quality of Service of the functional part the realization elements need to be Quality Aware (see ISO/IEC 23004-4 for more details). The resource management framework can manage all resource aware entities. This is not limited to M3W entities, but can also be other middleware entities or applications.

The Resource Management Framework (specification and realization) is depicted in Figure 6 — M3W resource management framework.

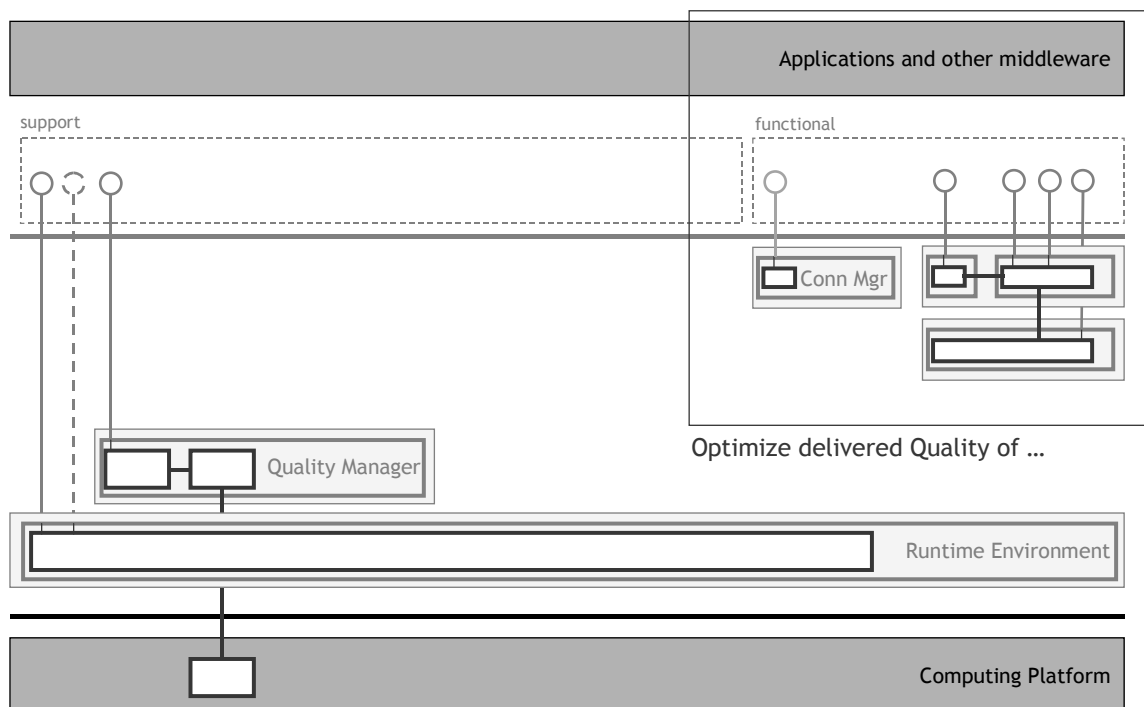


Figure 6 — M3W resource management framework

5.4.4 Download

The download framework is an optional addition to the M3W core framework. This framework enables the transfer (download as well as upload) of new components to permanent storage in the M3W system. The download framework consists of 5 roles:

- Target - enables receiving components
- Initiator - initiates and coordinates the download process
- Locator - responsible for locating all the entities (realizations of the roles) that participate in a particular download
- Decider - assesses the feasibility of a particular download.
- Repository - contains components that can be downloaded.

These roles can be deployed in a very flexible way in order to support different download and upload scenarios, varying from broadcast to on demand download for a specific M3W system. The only constraint is that the target has to be deployed on the M3W system to which the components need to be transferred.

The download framework (specification and realization) is depicted in Figure 7 — M3W download framework.

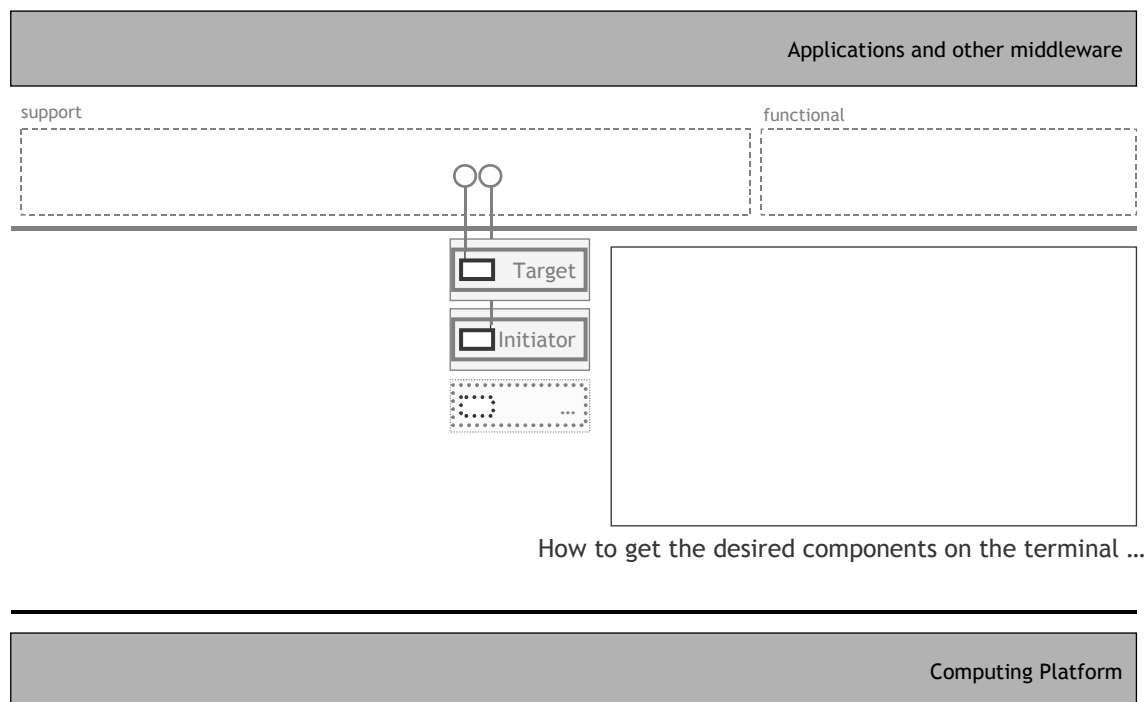


Figure 7 — M3W download framework

5.4.5 Fault Management

Fault Management framework is an optional additional framework to the M3W core framework. The Fault Management framework enables the transparent insertion of fault tolerance mechanisms in order to increase the dependability of an M3W system. In this context, 'transparent' means transparent for developers of M3W services (ISO/IEC 23004-3) as well as for the clients of these services.

The Fault Management framework enables the automatic generation of wrappers for M3W services. These wrappers contain fault tolerance mechanisms and aim to prevent that faults lead to failures of the M3W system. A wrapper with a fault tolerance mechanism is called a middleman.

When a client requests a service (a realization entity), then a middleman for this service can be instantiated automatically. More details can be found in ISO/IEC 23004-6.

The fault management framework is depicted in Figure 8 — M3W fault management framework.

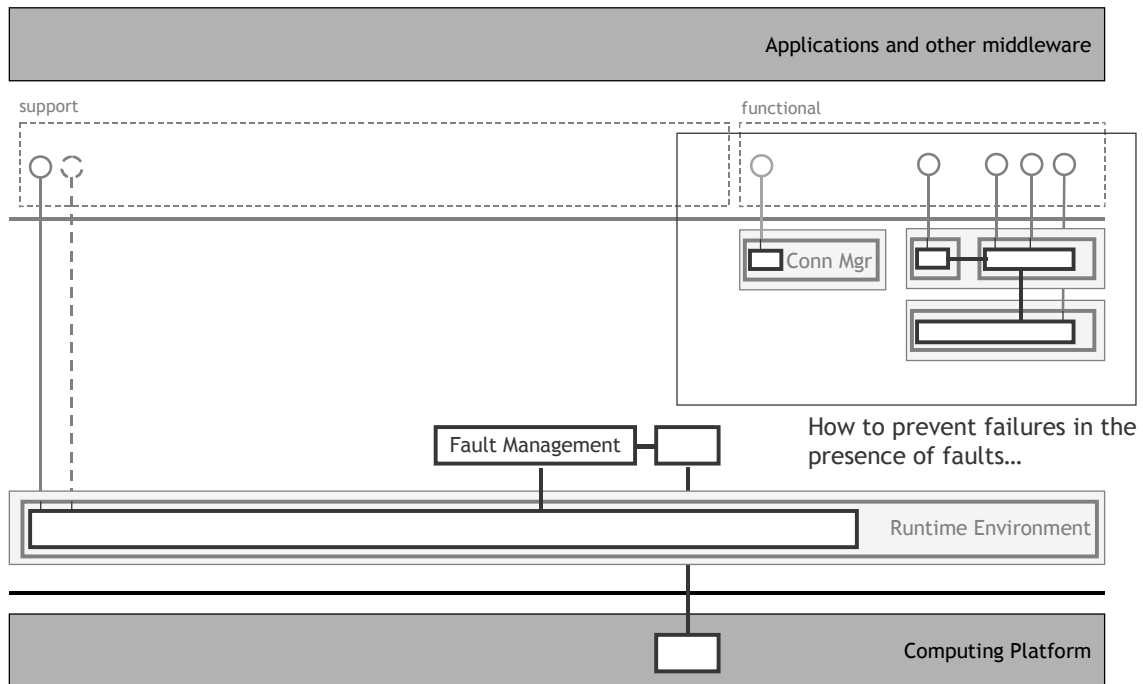


Figure 8 — M3W fault management framework

5.4.6 Integrity Management

Integrity Management is an optional addition to the M3W core framework. The Integrity Management framework enables the maintenance of a consistent software configuration for M3W. M3W systems have the ability to download and remove components, and also to register and un-register executable components. The Integrity Management framework enables the verification of the consistency of the current configuration, and when necessary, it can adapt the configuration in order to make the configuration consistent.

The Integrity Management framework focuses on the software configuration. It does not address run time errors, and recovery of these errors is the scope of the Fault Management framework (see subclause 5.4.5). Integrity Management does provide an escalation point for Fault Management. In the situation that it is confronted with a large number of runtime errors, then it can notify Integrity Management that verification of the consistency of the software configuration is desirable.

More information on the Integrity Management framework can be found in ISO/IEC 23004-7.

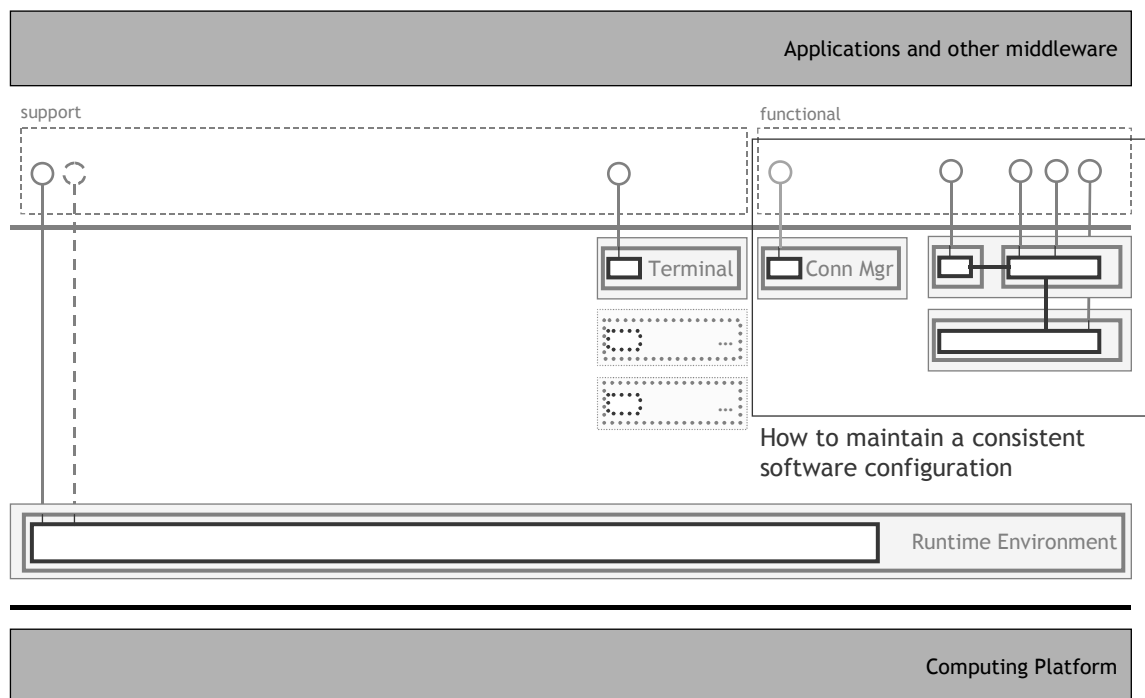


Figure 9 — M3W integrity management framework

5.5 Realization

M3W specifies an API and a realization technology. The API is specified in terms of logical components and roles (see Annex A). This specification gives an uniform view to M3W for developers of other middleware and applications. For the realization of M3W there is still enough implementation freedom to enable vendors to differentiate themselves.

A realization of M3W shall comply with:

- the API specification, and
- the realization technology.

The mapping from logical components and roles to realization elements is not specified in M3W. As long as the realization adheres to the constraints implied by the API specification and the realization technology, then it is OK.

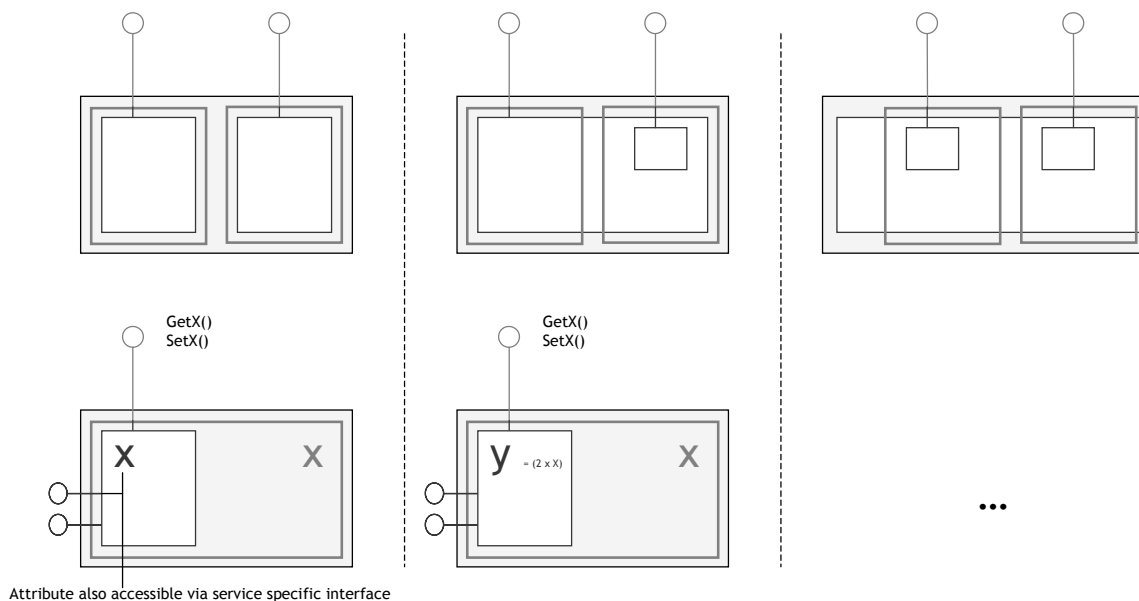


Figure 10 — Implementation freedom

EXAMPLE The mapping from components and roles to realization elements is not fixed. Multiple roles of a logical component can be realized by different services, multiple inner objects of a service, or a mixture (see ISO/IEC 23004-3 for more information on services and inner objects).

EXAMPLE The mapping from specification attributes on attributes in the realization elements is not fixed. An specification attribute can be mapped on an attribute of a realization element directly (same name and type), but also more indirectly (different name, type, etc.).

Annex A (informative)

API specifications reader's guide

A.1 The API specification

A.1.1 Introduction

The purpose of this subclause is to discuss some of the main concepts in connection with API specifications and to present the overall structure of an API specification.

A.1.2 Concepts

A.1.2.1 Interface Suites

The purpose of an API specification is to define a collection of software interfaces providing access to a coherent chunk of functionality. Such a collection of interfaces is referred to as an interface suite.

EXAMPLE The Analog Audio Decoding interface suite is a collection of interfaces providing access to Analog Audio Decoding functionality. Among other things, this interface suite contains an interface `uhIAAnaAdec` for controlling audio decoding and an interface `uhIAAnaAdecSelector` for controlling audio program selection.

A.1.2.2 Interfaces versus Functionality

The provision and use of interfaces should not be confused with the provision and use of functionality. Functionality is generally provided by means of a collection of interfaces (an interface suite). Generally speaking, the interfaces in this collection are 'provides' as well as 'requires' interfaces (from the point of view of the provider of the functionality).

EXAMPLE Providing Analog Audio Decoding functionality is not the same as providing the interfaces `uhIAAnaAdec` and `uhIAAnaAdecSelector`. The analog audio decoding and selection functionality is provided not only by means of the 'provides' interfaces `uhIAAnaAdec` and `uhIAAnaAdecSelector` but also by the 'requires' interface `uhIAAnaAdecNtf`, which should be provided by the client of the functionality.

A.1.2.3 Logical Components

From a conceptual point of view, the functionality associated with an interface suite can be viewed as a black box that interacts with its environment through the software interfaces in the interface suite and other interfaces such as streaming interfaces. Specifying the behavior associated with the interfaces amounts to specifying the external behavior of this black box. Because the black box is conceptual it is also referred to as a logical component.

Since interfaces and their associated functionality are inseparable, the terms interface suite and logical component are often used in an interchangeable way. In a narrow sense, the term logical component is sometimes reserved for those interface suites that are direct building blocks of the M3W API and not just building blocks of other interface suites. Examples of the latter are the `rcIUnknown` and Pin Objects interface suites and the generic Notification interface suite. These interface suites never occur stand-alone but always in combination with other interface suites.

EXAMPLE When using the terms Video Mixing logical component and Video Mixing interface suite we usually mean the same thing. The focus in the former is on the Video Mixing functionality while the focus in the latter is on the Video Mixing interfaces, but these two aspects are closely related.

A.1.2.4 Logical Components and Service

The concept of a logical component should not be confused with that of a realization element. An API specification specifies functionality and the way that this functionality can be accessed by means of interfaces. It does not specify how this functionality should be implemented, and it does not even specify how it should be packaged. The functionality associated with the interfaces may be implemented in a single realization element, but could just as well be implemented in multiple realization elements, as part of one or more realization elements.

EXAMPLE An implementation of the Video Mixing logical component is referred to as a video mixer. The video mixer need not be a single service but could be a collection of services.

A.1.2.5 API Specifications as Contracts

There are two sides to functionality: the side of the providers and the side of the clients of the functionality. Classically an API is viewed as a single interface that allows a client to access functionality implemented by a provider. The API specification can then be viewed as a bilateral contract between the provider and the client defining the rights and obligations of both. M3W API specifications are based on a generalized version of the classical contractual paradigm: an API specification is viewed as a multi-lateral contract defining multiple mutually related interfaces. The concept of role is key to this generalized notion of contract.

A.1.2.6 Roles

A role identifies a contractual party, where each role acts as a provider of one or more interfaces, as a client of one or more interfaces, or as both. Each role has contractual rights and obligations associated with it. When developing code that implements or uses the API, the developer should make explicit which roles are to be played by the code. The rights and obligations specified for these roles in the API specification then define which API-implied requirements have to be satisfied by the code.

EXAMPLE Three roles identified in the Analog Audio Decoding logical component are: *AnaAdec* (decoder), *AnaAdecSelector* (program selector) and *AnaAdecClient* (client of the interfaces). This illustrates that in the specification of a logical component the responsibilities for providing the overall functionality of the logical component may be divided over several roles played by the logical component.

A.1.2.7 Roles and Realization Elements

Roles should not be confused with realization elements (services) although the names of roles may sometimes suggest otherwise. Roles define required/allowed behavior and do not define how that behavior should be realized. The correspondence between roles and service need not be one-to-one. For example, a single role can be implemented by multiple services, and a single service class can implement multiple roles.

The UML term for a role is abstract class, i.e. a class representing behavior and having no implementation. This implies that roles are (abstract) classes and not objects. At run-time there may be multiple instances of a role, i.e. multiple objects that play the role.

EXAMPLE In an object-oriented implementation the *AnaAdecSelector* role from the Analog Audio Decoding logical component is typically implemented by a co-class with the same name. Because an analog audio decoder/selector has multiple program selectors, multiple objects of type *AnaAdecSelector* will be created at run-time, each of which can be viewed as an instance of the *AnaAdecSelector* role.

A.1.3 Patterns

An API specification is itself a composition of more fine-grained specifications which define specification items such as constants, data types, roles, interfaces and methods. For each type of specification item fixed specification patterns are used, and these are discussed in the following subclauses. Although different, all of these specification patterns have a common super-structure defined by a number of sections that occur in all specification patterns (and in fixed order). Being aware of this common structure makes it easier to understand the specifications. The common subclauses are:

1) Signature

Contains the IDL definition of the specification item. The text in this subclause occurs literally in the IDL definition of the API as a whole. The only exception is the Signature subclause of a model type or constant, which consists of pseudo-IDL that will not be contained in the IDL of the API. Interface specifications do not have a Signature subclause because the contents of this subclause are essentially only the concatenation of the Signature subclauses of the methods defined in the interface.

2) Qualifiers

Provides a list of qualifiers which are predefined keywords that can be associated with a specification item. Each qualifier represents a property or constraint that is being imposed on the specification item. The valid qualifiers for each specification item and their meaning are defined in Annex J, API Qualifiers. Qualifiers are similar to stereotypes in UML except that multiple qualifiers may be associated with a specification item. They are typically used as abbreviations for standard execution constraints (e.g. the qualifier single-threaded) or the specification of standard behavior (e.g. the qualifier subscribe-function).

EXAMPLE Three roles identified in the Analog Audio Decoding logical component are:

- `AnaAdec` (decoder),
- `AnaAdecSelector` (program selector) and
- `AnaAdecClient` (client of the interfaces).

These illustrate the use of the following qualifiers:

- `Ana` (for analog), and
- `Adec` (for audio-decoder).

3) Description

Provides a short informal description of the specification item.

4) (Execution) Constraints

Defines restrictions that apply to the specification item.

Examples of these are restrictions on the set of allowed values of a data type (e.g. min-max constraints) and execution constraints that impose multi-threading constraints on the clients of interfaces. Note that several frequently occurring constraints (such as being single-threaded) have been defined as qualifiers, so these constraints are specified in the Qualifiers subclause rather than the Constraints subclause.

5) Remarks

Contains a list of remarks related to the specification item.

A.1.4 Structure of an API Specification

The idea of API specifications as contracts is reflected in the structure of an API specification which consists of the following subclauses:

1) Concepts

This subclause introduces the concepts associated with the functionality being specified and to define the vocabulary used in formulating the contractual rights and obligations.

2) Types & Constants

This subclause defines types and constants that are used in the specification. Typical content would include: the definitions of data types that occur in the parameter lists of interface functions and the definitions of the specific error codes that can be returned by these functions.

3) Logical Component

This subclause defines overall aspects of the logical component such as its structure, its diversity and its instantiation. A central position in this subclause is taken by the “interface-role model” which is a UML class diagram showing all roles and interfaces and their mutual relations.

4) Roles

This subclause specifies the roles that represent the functionality provided by the API. A ‘model-oriented’ style of specification is used, where attributes are used to represent state information associated with the roles. Interface functions operate using these attributes but roles can also modify them autonomously.

5) Interfaces

This subclause contains the actual interface specifications, in particular the specifications of the functions that occur in the interfaces. These functions define the part of role behavior that can be controlled externally from the software. The common style of specification is (extended) pre- and postconditions.

6) Additional subclauses

Used to collect any relevant information that does not fit into any of the other subclauses.

The structure and contents of the above subclauses are discussed in more detail in the following subclauses.

A.2 The ‘Concepts’ subclause

A.2.1 General

The Concepts subclause introduces and discusses the functionality defined in the API specification in an informal and intuitive way. It explains the main terms and concepts that play a role, and indicates the context in which the functionality will be used. The concepts are typically illustrated using pictures and diagrams. The contents of this subclause are free format, and so the subclause has no predefined structure.

A.3 The 'Types & Constants' subclause

A.3.1 General

The Types & Constants subclause contains the specifications of all types and constants that play a role in the API specification, and that are not already specified in one of the base documents. The subclause is divided into two parts.

The first part deals with the public types and constants which are the types and constants that are visible to the application programmer: their definitions occur in the IDL associated with the API.

The second part deals with the model types and constants which are the types and constants introduced for modeling purposes only: these types and constants will not occur in the IDL of the API.

Public and model types and constants are specified in essentially the same way except that the definitions of model types and constants can use additional data types. Each type is specified in a separate subclause; the name of the type is used as the heading of this subclause. Type specifications are discussed in A.3.3 and A.3.4.

Constants are specified in groups of related constants. Each group is defined in a separate subclause; the name of the group (e.g. Error Codes) is used as the heading of this subclause. In the case where a constant group consists of a single constant, then the name of the constant is used as the name of the constant group. Constant specifications are discussed in A.3.2.

A.3.2 Constant Specifications

A constant specification defines a group of related constants and consists of the following subclauses, some of which are optional:

1) Signature

The IDL definition of the constants. In the case of model constants this may be pseudo-IDL.

2) Qualifiers

The qualifiers associated with the constants. An example is the error-codes qualifier which states that the constants being defined are error codes returned by functions. If there are no qualifiers then this subclause will contain the word; 'none'.

3) Description

A short informal description of the group of constants.

4) Constants (optional)

Contains a table providing a short informal description of each constant in the group. In case the group of constants being defined consists of a single constant this subclause is omitted and the informal description of the constant is contained in the Description subclause.

5) Remarks (optional)

Contains a list of remarks related to the constants.

A.3.3 Type Specifications

A type specification defines a data type and consists of the following subclauses, some of which are optional:

1) Signature

The IDL definition of the data type. In the case of a model type this may be pseudo-IDL.

2) Qualifiers

The qualifiers associated with the type. Examples of frequently used qualifiers are the enum-element and enum-set qualifiers for enumerated data types; these are explained in A.3.4.

3) Description

A short informal description of the data type.

4) Values / Fields / Attributes

Contains a table providing a short informal description of each member of the data type. In the case of an enumerated type the members are the values being enumerated. In the case of a structure type the members are the fields of the structure. Entity types are modeling types that have attributes as members; they are discussed in A.3.5. The heading of this subclause (if present) varies dependent on the kind of data type being defined.

5) Constraints

Defines restrictions that apply to the values of the data type. For example, when defining a structure type there could be a dependency between the values of the fields which can be expressed as a constraint.

6) Remarks (optional)

Contains a list of remarks related to the data type.

A.3.4 Enums and Enum Sets

Enumerated data types are frequently used in APIs. It is also common practice in C to use bitwise or-ing of enumerated values to represent sets of enumerated values. In common programming practice no distinction is made between the enumerated type and the set type which can lead to confusion. In API specifications the distinction between the two is explicitly made. The qualifiers enum-element and enum-set are used for this.

If the values of an enumerated type are meant to be used as elements of sets (bit vectors), the enumerated type gets the qualifier enum-element. In that case the values of the enumerated type are defined as powers of two.

Data types that represent sets of enumerated values are typedef-ed as UInt32 (32-bit bit vectors) and get the qualifier enum-set. Normal enumerated types whose values are not meant to be bitwise or-ed are defined as usual, without using a qualifier.

A.3.5 Model Types

Besides the normal data types that can occur in IDL, such as Int32, Bool, enum{...}, struct{...}, etc., M3W API specifications can also contain abstract data types that are used for specification purposes only. These data types are defined in the Model Types & Constants subclause of the Types & Constants subclause. The naming of model types is more liberal than that of public types and constants. Names of model types usually have no prefixes and suffixes such as `uh` and `_t`.

Model types are mainly used in the definitions of attributes associated with roles. Besides the normal IDL data types the following three modeling types are used in API specifications:

1) Set

A set type defines (possibly infinite) mathematical sets of values of a specific type. The notation `setof{type}` is used to denote the type of mathematical sets of values of type 'type'. For sets the usual mathematical notations are used.

2) Map

A map type defines tables that map values of one type (the domain type) to values of another type (the range type). Values of this type, called maps, can be viewed as generalized arrays and identical notation is used for these types. An array can be viewed as a map whose domain type is a sub range of the integers starting at 0; in maps the domain and range types may be any type.

3) Entity

An entity type defines a collection of objects that may have attributes associated with them. In UML terms, an entity type is a class with public attributes only and no operations, and an entity is an instance of the class, i.e. an object. The notation `entity{...}` is used to declare an entity type where the attributes are declared as C-style variables inside the braces.

The notations and the specification pattern used for entities are similar to those used for roles (see A.5). Similar to the specification of role attributes, the attributes of an entity type are divided into independent attributes, whose value may in principle vary freely, and dependent attributes, whose value can be defined in terms of other values. An entity type, like a role, can also be defined as a specialization of another entity type.

A.4 The 'Logical Component' subclause

A.4.1 General

The Logical Component subclause specifies all aspects of the API functionality that relate to the logical component as a whole, i.e. those aspects that are not specific to a particular role, interface or function. This subclause consists of four subclauses:

1) Interface-Role Model

Defines the interface-role model which can be viewed as a graphical summary of the main contractual entities, in particular, the roles and interfaces and their mutual relations.

2) Diversity

Defines the parameters that can be set at instantiation time of the logical component.

3) Instantiation

Specifies the result of instantiating a logical component: the objects that are created and their initial state.

4) Execution Constraints

Defines concurrency-related constraints that apply to the logical component as a whole.

These subclauses are discussed in more detail below.

A.4.2 Interface-Role Model

One of the explicit goals of the style of specification used in the API specifications is to improve the level of precision of the API specifications without becoming too formal. Besides using a contractual approach, this goal can be achieved by using a model-oriented style of specification based on a restricted use of UML. Model-oriented specifications define behavior in terms of a model which can be seen as a kind of abstract implementation. The interface-role model defined in the Interface-Role Model subclause is an enhanced UML class diagram that acts as a graphical summary of the model used to specify the logical component. It shows the main entities in the model such as interfaces and roles while leaving out other entities such as attributes and constraints. (Constraints are dealt with elsewhere (in a non-graphical way)). Each API specification defines such an interface-role model.

The interface-role model consists of the following entities:

- 1) The interfaces defined in the API specification possibly including interfaces defined in the base documents. The interfaces are represented by 'lollipops'.



- 2) The roles defined in the API specification possibly including roles defined in the base documents. The roles are represented as (abstract) UML classes.
- 3) The 'provides' and 'requires' (or use) relations between the roles and interfaces. The 'provides' relation is represented by the stick of a lollipop and the 'requires' relation by a UML dependency relation (a dotted arrow).



- 4) The specializes relation between roles, represented by the inverse UML generalization arrow. Role R2 specializing role R1 implies that all contractual rights and obligations that apply to R1 also apply to R2.
- 5) The streams associated with the roles. Streams are represented by small squares.



- 6) The input and output relations between roles and streams. The input relation is represented by a fat arrow connecting a stream and a role and the output relation by a fat line connecting a role and a stream.
- 7) The bounding box of the logical component represented by a grey box. What is inside the box is considered internal to the logical component and what is outside is considered external.

Figure A.1 — Interface Role Model: Notations Used, shows an example of the use of an Interface Role Model. It provides an overview of the UML notations used in interface-role diagrams. The small white squares are classes representing streams. A stream may be associated with a role as an input, an output, or both, as indicated by the fat lines and arrows. The small white squares outside the grey box represent the external input and output streams and those inside the grey box represent internal streams introduced for modeling

purposes only. In practice, the small white squares representing the streams are sometimes omitted and only the fat lines and arrows are shown. Note that the specialization relation, although shown as a relation between two client roles, will most often occur as a relation between the roles inside the grey box.

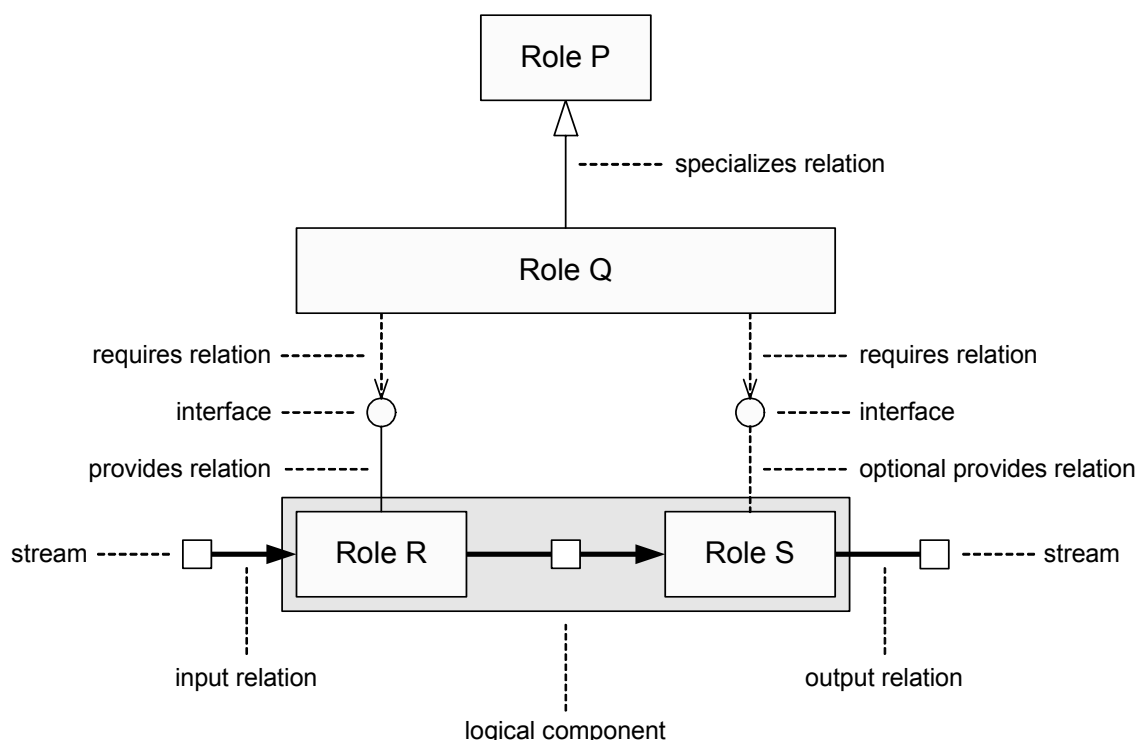


Figure A.1 — Interface Role Model: Notations Used

Roles, interfaces and streams defined in base documents are not normally shown in the interface-role diagram except when they relate to roles that are specialized by roles defined in the API specification. The roles and interfaces defined in the rclUnknown and Notification API specifications are exceptions in that they are not normally shown in the diagram even when being specialized.

Note that the interface-role model provides information on the navigability between the interfaces of the logical component. If two interfaces are provided (directly or indirectly) by the same role, the QueryInterface function of rclUnknown can be used to navigate from one interface to another. If two interfaces are provided by two different roles this is not possible.

A.4.3 Diversity

The Diversity subclause gives a survey of the diversity associated with the logical component being defined. The diversity of a logical component consists of all parameters that can be set at instantiation time of the logical component and that will not change during the lifetime of the logical component. It is divided into interface diversity, defining which interfaces are mandatory/optional when instantiating an instance of the logical component, and configuration diversity, defining other parameters that can be set at instantiation time. The latter are modelled by attributes of roles. The interface and configuration diversity are specified in separate subclauses. There is a third subclause which is used for specifying constraints on the diversity parameters.

A.4.3.1 Provided Interfaces

The Provided Interfaces subclause contains a table defining, for each interface of the API, whether that interface is mandatory or optional. The Role column in this table, indicating the provider of an interface, is

necessary because the same interface may occur on multiple roles, in particular when two roles specialize the same role defined in a base document.

Note that an interface IR being mandatory on a role R means that each instance of R has the obligation to provide IR. An interface IR being optional on a role R means that each instance of R has the right to provide IR but not the obligation. Note that if IR is optional on R, it could be mandatory on a role S that specializes R but not the other way around. At runtime the presence or absence of an interface can be checked by means of the `QueryInterface` function of the omnipresent `rcIUnknown`.

A.4.3.2 Configurable Items

The Configurable Items subclause contains a table defining the configuration parameters that can be set when the logical component is instantiated. The parameters are modeled as role attributes (diversity attributes) that are defined in the Roles subclause. Attributes referred to in this table may be constant as well as variable. If the attribute is variable, the value that is provided at instantiation time is interpreted as the initial value of the attribute. So, the initial value of the attribute is the configuration parameter, and not its current value (which may change).

Note that API specifications only specify which parameters can be set at instantiation time and not how they are set (e.g. by compile-time variables, properties in a database, parameters of an instantiation function, etc.). The mechanism used to set the configuration parameters is implementation-dependent.

A.4.3.3 Constraints

The Constraints subclause specifies constraints that apply to the optional interfaces and configurable items. These constraints can be viewed as the precondition of the instantiation operation.

A.4.4 Instantiation

Instantiation refers to the process of creating instances of logical components. A logical component instance is not a single object (in the object-oriented sense), but an aggregate that may consist of several objects. The API specification makes no assumptions on the mechanisms used to create a logical component instance but specifies what the result of the instantiation of a logical component is. This result is specified in the Instantiation subclause which consists of two subclauses, one specifying which objects are created, and one specifying the initial state of these objects.

A.4.4.1 Objects Created

The Objects Created subclause defines which objects are contained in a new instance of the logical component. This is done by means of a table that specifies for each object: its type (normally a role name); its name; and its multiplicity. The multiplicity n of an object x indicates how many instances of the object occur in the logical component instance. If $n > 1$ then x is interpreted as an array of objects of size n . The individual objects in the array are indicated as usual: $x[0]$, $x[1]$, ..., $x[n - 1]$. The entries in the Object column in the table act as global names that may be referred to in the rest of the specification.

A.4.4.2 Initial State

The Initial State subclause specifies properties of the objects declared in the previous subclause that are true in the initial state of the logical component instance, i.e. immediately after the instantiation of the logical component. Properties that follow from the diversity constraints are not repeated here, so that information should be added, in order to get the full specification of the initial state of the logical component instance. Note that certain aspects of the initial state may have been deliberately left unspecified, e.g. to create implementation freedom.

A.4.5 Execution Constraints

In the Execution Constraints subclause, the execution constraints that apply to the logical component as a whole are specified. In particular, global constraints with respect to the multi-threading aspects of the logical

component are specified here. For example, a constraint might be that clients may not concurrently access any function in any interface provided by any role that is part of the logical component. These constraints are not repeated in the other subclauses that follow.

A.5 The 'Roles' subclause

A.5.1 General

The Roles subclause specifies the roles used in the API specification. Each role is specified in a separate subclause that has the name of the role as its title. A role represents behavior associated with one or more interfaces. Which ('provides' and 'requires') interfaces are associated with a role can be seen in the interface-role model.

Roles are a means to define the observable behavior of the logical component, i.e. the behavior that can be observed at the external software and streaming interfaces of the logical component. What is observable and what is not, are defined by the boundaries of the logical component box in the interface-role diagram.

The behavior associated with a role can be divided into four aspects:

- 1) Function behavior: the behavior of the functions in the interfaces provided by the role.
- 2) Streaming behavior: the input-output behavior of the streams associated with the role.
- 3) Active behavior: autonomous behavior that is visible at the 'provides' and 'requires' interfaces associated with the role.
- 4) Instantiation behavior: behavior displayed at instantiation time of the logical component.

Only the streaming and active behavior are specified in the Roles subclause. Instantiation behavior is specified in the Logical Component subclause and function behavior is specified in the Interfaces subclause; see Figure A.2 — Role Behavior and the Subclauses Where it is Specified.

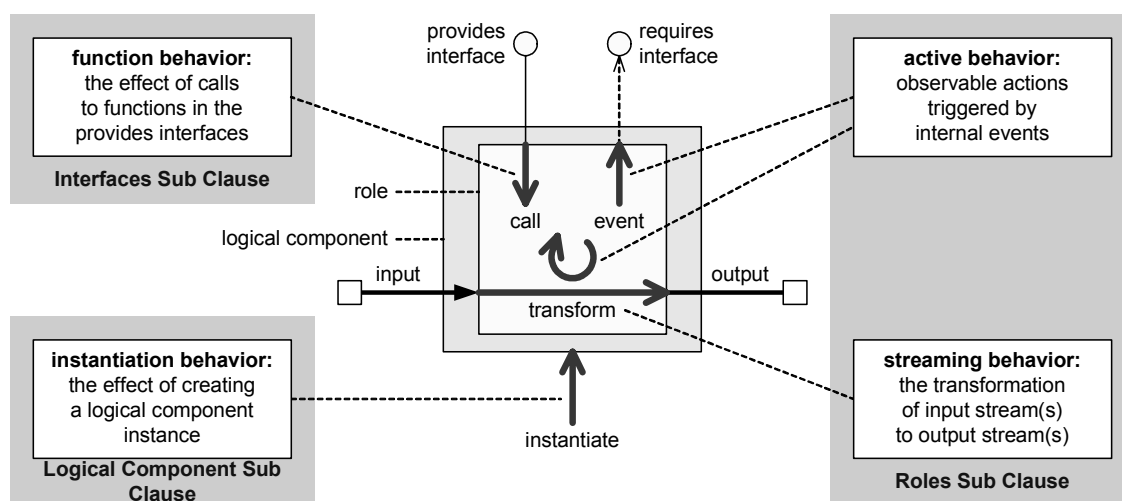


Figure A.2 — Role Behavior and the Subclauses Where it is Specified

NOTE Roles are pure specification artefacts. When implementing a logical component the only requirement is that the observable behavior of the logical component is consistent with the observable role behavior.

NOTE Role behavior is realized dynamically by instances of the role, i.e. by objects playing the role. So a role should not itself be viewed as an object, but rather as a class that may have several instances at run-time. From a contractual point of view, the specification of the role can be viewed as the set of rights and obligations to be satisfied by the code of that class.

A.5.2 Role Specifications

A role specification consists of the following subclauses, some of which are optional:

1) Signature

The pseudo-IDL definition of the role which defines the attributes associated with the role. Attributes can be viewed as variables representing the abstract state of a role instance. They play an essential role in the model-oriented style of specification used in the API specifications.

2) Qualifiers

The qualifiers associated with the role.

3) Description

A short informal description of the role.

4) Independent Attributes

Contains a table providing a short informal description of each independent attribute. An independent attribute is an attribute whose value may in principle vary freely.

5) Dependent Attributes

Contains a table providing a short informal description of each dependent attribute. A dependent attribute is an attribute whose value can be expressed in terms of other entities, in particular the values of independent attributes. The description of the dependent attribute includes a definition of the value of the attribute.

6) Invariants

Provides a list of invariants. An invariant is an assertion about the role that is always true from the external observer's point of view. The assertion is typically formulated in terms of the attributes of the role. In reality, the assertion may temporarily be violated.

7) Instantiation

Provides a description of how instances of the role are created. Usually instances of a role are created at instantiation time of a logical component only. In some cases instances of a role can also be created dynamically after the instantiation of the logical component, in particular by using interface functions that act as "constructors" of the role. In that case a list of the names of these constructors is provided.

8) Streaming Behavior

A description of the streaming behavior of the role, i.e. the input-output behavior of the streams associated with the role.

9) Active Behavior

A description of the active behavior of the role, i.e. autonomous behavior that is visible at the interfaces associated with the role.

10) Execution Constraints

Defines concurrency-related constraints that apply to the role, in particular to the collection of interfaces associated with the role. These constraints are not repeated in the specifications of these interfaces.

11) Remarks (optional)

Contains a list of remarks related to the role.

A.5.3 Role Signatures

The role signature can be viewed as the IDL of the role. It defines the structure of a role in the same way that IDL defines the structure of an interface. More specifically, the role signature defines which roles are specialized by the role and which attributes are associated with the role. From a contractual point of view a role S specializing a role R amounts to S inheriting all rights and obligations specified for R, including all attributes of R.

Attributes can be viewed as variables representing the abstract state of a role instance. The behavior of the logical component (see Figure A.2 — Role Behavior and the Subclauses Where it is Specified) is specified in terms of this abstract state. Typically, external calls of interface functions will change the values of attributes and thereby influence the streaming and active behavior of a role. Conversely, the values of attributes may change autonomously as a consequence of the streaming and active behavior, which in turn may influence the behavior of other roles.

Attributes are introduced for modeling purposes only and should not be confused with implementation variables. They correspond to UML attributes but are not indicated in the interface-role model (which is an extended UML class diagram).

Note that attributes inherited from base roles (roles defined in base documents) are not re-declared in the signature. The optional `const` keyword indicates that the value of an attribute is constant during the lifetime of a role instance. It does not imply that the value of the attribute is the same for different role instances.

A.5.4 Independent and Dependent Attributes

The attributes associated with a role are divided into two types: those whose value may be defined or changed independently of all other attributes, and those whose value is a function of the values of other attributes. These two types are referred to as independent and dependent attributes. The difference makes sense because when specifying state transitions, only the changes to the values of the independent attributes have to be specified; the values of the dependent attributes follow automatically.

Independent attributes can be viewed as variables whose value can be modified. The independent attributes are listed in a table containing the name of each independent attribute and a short description of the meaning of the attribute.

The value of a dependent attribute can be expressed in terms of other entities, in particular the values of independent attributes. Dependent attributes can be viewed as parameterless functions whose value can be computed.

Dependent attributes are listed in a separate table similar to the dependent attributes. The difference is that the table also defines the value of each attribute (as part of the description of the attribute).

Sometimes it is known that an attribute is dependent on other entities without knowing what the exact dependencies are. This happens particularly with attributes that model signal properties; their value depends on the contents of the signal. In these cases the value of the attribute is only defined informally.

Dependent attributes are sometimes called auxiliary or convenience attributes. We can, in principle, do without them by repeating the expression defining the value of a dependent attribute every time we have to

refer to that value, but the appropriate use of dependent attributes has a positive effect on both the size and the maintainability of an API specification.

A.5.5 Streaming Behavior

API specifications are meant to specify software interfaces, i.e. interfaces that can be used by software developers to develop applications on top of a platform. Many of the M3W API interfaces are streaming-related; they are used to control or provide information about streaming functionality. There is no way such interfaces can be defined without referring to streaming functionality. On the other hand, it is not the purpose of the M3W API specifications to provide detailed specifications of the streaming itself. The API specifications define software interfaces that are common to all platform instances. In a concrete platform instance, these specifications are augmented with additional information concerning performance, resource usage, streaming algorithms used, etc.

The purpose of the Streaming Behavior subclause is to define the streaming behavior associated with the role at a level of abstraction that is sufficiently concrete to define the effect of the functions in the interfaces, and that is sufficiently abstract to make the interface generally applicable to different platform instances. In certain special cases this may mean that streaming behavior is specified in detail, but in most cases streaming behavior is modeled abstractly with a reference to standards for further details. The description of streaming behavior starts in the interface-role model diagram where the input and output streams are indicated, together with possible internal streams. The input and output streams are weakly typed. The description of how an input stream is transformed into an output stream is generally informal, where role attributes may be used to represent stream content.

A.5.6 Active Behavior

Besides autonomously performing streaming functions, instances of roles may also autonomously perform software functions. A typical case is the notification of events detected in input streams. Autonomous behavior is often specified in terms of an event-action table that connects events that may occur to actions performed by a role instance. The events can be streaming-related events but also software-related events such as timer events. The actions can be calls to functions in 'requires' interfaces or modifications of attributes. Roles which have no active behavior are passive; they do not actively influence the behavior of other roles.

Note that even though event notification is conceptually part of the active behavior associated with a role, most event notifications supported by a role are not listed in the event-action table. The reason is that the most common form of event behavior, an event leading to a notification of all subscribers of the event, is already specified in the general notification specification. This behavior is not repeated for every single event in the event-action table. The description of the event and the conditions under which the event is raised are described in the specification of the corresponding callback function. Some events are special in that they may have additional effects besides notifying clients. Only these non-standard event notifications, together with other autonomous behavior of role instances, are specified in the Active Behavior subclause.

A.5.7 Actor Roles

In the interface-role model of a logical component, roles are introduced not only for the objects that provide the functionality of the logical component, but also for the objects that use the functionality, i.e. the clients of the logical component. The reason is that, from the contractual point of view, not only the providers, but also the users of the logical component may have to satisfy certain obligations. The latter obligations are part of the contract, and are associated with client roles that model the users of the logical component. Examples of client obligations are that a client has to provide a notification interface, or that it should not call certain functions under certain conditions.

From the specification point of view, client roles are usually much simpler than the provider roles of a logical component. A client role is typically characterized by having no attributes and no streaming behavior while its active behavior consists of providing stimuli only, i.e. of calling functions in interfaces of the logical component (as indicated in the interface-role model), without any a priori assumptions on when these calls occur. The qualifier actor is used to indicate this type of role and the streaming and active behavior as well as the attribute subclauses are omitted in the specification. Furthermore, no assumptions are made on how instances of an actor role are created.

A.6 The 'Interfaces' subclause

A.6.1 General

The Interfaces subclause contains specifications of all interfaces that are part of the API. The bulk of the specification of an interface consists of the specifications of the individual functions that constitute the interface. Each interface is specified in a separate subclause that has the name of the interface as its title. The structure and contents of an interface specification are discussed in A.6.2.

Each function in an interface is specified in a separate subclause of the interface specification. The structure and contents of a function specification are discussed in A.6.3. The style of specification used is extended pre- and postconditions which is further explained in A.6.4.

There are two types of interfaces: control interfaces and notification interfaces, which are dealt with slightly differently at the specification level. The differences between these two types of interface specifications are discussed in A.6.5.

Some frequently occurring types of functions are dealt with in a special way. This includes some standard functions relating to event subscription (discussed in A.6.6) and asynchronous functions (discussed in A.6.7). Finally, in A.6.8 we explain the "bullet notation" that is frequently used in function specifications.

A.6.2 Interface Specifications

An interface specification defines either a new interface or an interface that is specialized by the logical component. In the case of a new interface, the name of the subclause defining the interface is equal to the unqualified interface name, e.g. `uhIAnaAdec`. In the case of a specialized interface, the name of the role providing the interface is added as a prefix to the interface name, using "::" as a separator. For example, "`AnaAdec::uhIPinObjects`" can be read as "the interface `uhIPinObjects` as specialized by the role `AnaAdec`". The reason for this convention is that there may be multiple roles that specialize the same interface. Each of these specializations is described in a separate subclause.

An interface specification consists of the following subclauses, some of which are optional (which?).

1) Qualifiers

The qualifiers associated with the interface. An example of a frequently used qualifier is `callback` which marks the interface as a callback interface; the typical example of a callback interface is a notification interface.

2) Description

A short informal description of the interface.

3) Interface ID

The globally unique identifier associated with the interface.

4) Execution Constraints

Defines concurrency-related constraints that apply to the interface, in particular to the collection of functions in the interface. These constraints are not repeated in the specifications of these functions.

5) Remarks

Contains a list of remarks related to the interface.

6) Function specifications

Specifications of the individual functions in the interface, each in a separate subclause. The standard functions inherited from the `rcIUnknown` interface (`QueryInterface`, `AddRef` and `Release`) are omitted.

There is no separate subclause defining the signature (IDL representation) of the interface, because the signature can be generated automatically from the name of the interface, the interface ID and the signatures of the functions in the interface.

The above template is used for new as well as specialized interfaces. The only difference for specialized interfaces is that there are no specifications for those functions that are inherited as is from the base interface. Only those functions whose specification has changed by the specialization get a separate specification that overrides the old specification.

A.6.3 Function Specifications

Except for one detail (explained in A.6.4), the structure of a function specification is fairly standard. The style of specification can be characterized as extended pre- and postconditions. The specification of a function is subdivided into the following subclauses, some of which are optional (which?):

1) Signature

The IDL definition of the function. This is essentially the C prototype of the function with some extra information, e.g. indicating which parameters are in-parameters and which ones are out-parameters.

2) Qualifiers

The qualifiers associated with the function. These qualifiers are often execution-related. For example, the “single-threaded” qualifier indicates that the number of threads that may concurrently execute the method is at most one.

3) Description

A short informal description of the function.

4) Parameters

Short informal descriptions of the parameters of the function, arranged in a table.

5) Return values

Short informal descriptions of the values that can be returned by the function, arranged in a table. In M3W API control interfaces the return values are almost always error codes; functions in notification functions do not normally return values. If the function returns only standard error codes, the table is omitted and the keyword “Standard” is used instead.

6) Precondition

Defines an assertion that should be true immediately before execution of the function starts. Contractually speaking, it is the obligation of the caller of the function to make sure this is the case.

7) Action

Describes the “abstract action” performed by the callee when the function is called. This action typically indicates which role attributes are modified by the function and/or which out-calls are performed by the function.

8) Postcondition

Defines an assertion that is true immediately after execution of the function finishes. Contractually speaking, it is the obligation of the callee to make sure this is the case.

9) Remarks (optional)

Contains a list of remarks related to the function.

A.6.4 Preconditions, Actions and Postconditions

An inherent restriction of classical pre- and postcondition specifications is that they can only be used to specify constraints on the states, before and after the execution of a function. They cannot be used to specify constraints on what should happen between these two states, i.e. during the execution of the function. A common approach is to allow anything to happen during the execution as long as the postcondition is not violated. This approach leads to underspecification of functions because we normally do not want a function to modify arbitrary variables or call arbitrary functions during its execution, even if the postcondition is met in the end. Furthermore, we sometimes want to express that certain observable actions should occur during the execution of a function, which is hard or even impossible to express in a postcondition. Typical examples of such actions are out-calls (such as callbacks) and synchronization actions (blocking).

The above problem is solved in API specifications by using extended pre- and postcondition specifications that allow observable behavior during the execution of a function to be specified in terms of an abstract action. The action is described in a way similar to ordinary code but using more abstract non-deterministic constructs that leave a lot of implementation freedom. The action is generally simple, though it can be used to specify more complex behavior as well, such as out-calls and synchronization.

A typical construct used in formulating abstract actions is Modify *x*, where *x* is some attribute (see the example above). Modify *x* should be interpreted as “change the state of the logical component in such a way that no state variable other than *x* is modified”. The new value of *x* is typically specified in the postcondition, where we do not have to specify that other variables have not been modified. In order to refer to the old value of *x* in the postcondition, the notation *x'* is used. So, in the example above, when the notation *mixer.blanked'* would have been used in the postcondition, it would refer to the value of *mixer.blanked* immediately before the call of the `BlankOutput` function. The Modify clause for an out-parameter of a function is omitted because the right to modify the value of the parameter is implicit in the [out] attribute associated with the parameter.

Each extended pre- and postcondition specification of a function can be interpreted as a mini-contract between the caller and callee of the function, as indicated in Figure A.3 — Contractual Interpretation of Extended Pre- and Postcondition Specifications.

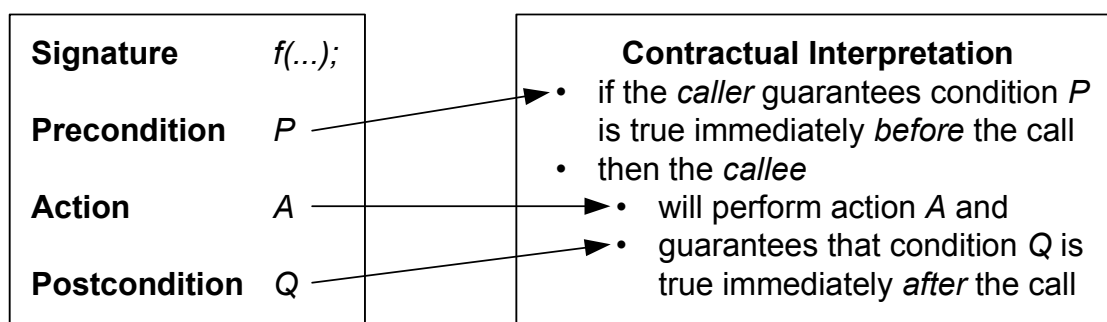


Figure A.3 — Contractual Interpretation of Extended Pre- and Postcondition Specifications

A.6.5 Control and Notification Interfaces

Generally speaking, a logical component provides two types of functionality: (streaming) control and (event) notification. The control functionality is provided by means of one or more control interfaces that are provided by the logical component and that allow a client to influence the behavior of the logical component by calling functions. The notification functionality is provided by one or more notification interfaces that are provided by the client and that allow the logical component to notify the occurrence of events to the client by calling functions.

Although the same layout is used for defining control and notification interfaces, they are dealt with differently at the specification level. The main reason for this is that, from the point of view of a logical component, a control interface is a 'provides' interface and a notification interface is a 'requires' interface. The specification of a control interface will define precisely what the effect of each function in the interface is. The specification of a notification interface, on the other hand, will usually not define what the effect of a notification function is; that is up to the client implementing the interface. Instead, the specification will describe when the function will be called, i.e. which event will cause the function to be called by the logical component.

Notification interfaces can be recognized by their name ending in `Notify`. Typically, the name of a notification interface is equal to the name of the corresponding control interface extended with `Notify`.

A.6.6 Subscribe, Unsubscribe and OnSubscriptionChanged

In order to make a logical component notify the occurrence of an event to a client, the client should first subscribe to the event. Each notification interface therefore has an associated control interface that among other functions contains two standard functions `Subscribe` and `Unsubscribe`. There is also one standard function `OnSubscriptionChanged` in each notification interface which is used to notify the client that its subscription has changed (because subscription is asynchronous). Because the specifications of these three functions have a standard pattern, their specifications are abbreviated using special qualifiers.

A.6.7 Asynchronous Functions

An asynchronous function can be viewed as a function with a delayed effect. The behavior of such a function can be separated into a synchronous part, i.e. the behavior between call and return of the function, and an asynchronous part, i.e. the behavior after returning from the function. Completion of the asynchronous action is usually reported by means of a notification allowing the caller of the function to synchronize with completion of the asynchronous effect of the function.

Because asynchronous functions occur frequently, a special extension of the pre-action-post format is used for specifying them. The synchronous part of an asynchronous function is specified in the same way as a synchronous function (except for the qualifier `asynchronous`). The asynchronous part is specified by means of an additional action-postcondition pair specifying the effect of the asynchronous action performed by the function.

The following notation is used in the Asynchronous Action clause to indicate the (asynchronous) act of notifying all subscribers to a specific event:

```
Notify OnEvent(*, ...)
```

The asterisk refers to the observer-specific cookie and the ellipses to the event-specific data that is being passed.

The absence of the synchronous Action and Postcondition clauses in an asynchronous function specification are equivalent to the synchronous action being `None` and the synchronous postcondition being `True`. In this case the function does not have an immediate effect, but a delayed effect only.

The asynchronous action that is the consequence of calling an asynchronous function is part of the active behavior of the logical component. Because this action is already specified in the asynchronous function

specification, its specification is not repeated in the specification of the active behavior of the role that provides the function.

A.6.8 Bullet Notation

Throughout the API specification bullets and indentation are used to structure possibly complex assertions and expressions and restrict the number of parentheses as much as possible. The items in a bullet list that represents an assertion (such as an invariant, precondition or postcondition) are conceptually connected by logical AND operators. The items in a bullet list that represents an action are conceptually connected by sequential composition operators (semicolons, in C terms). Start and end of indentation logically introduce an opening and closing bracket, respectively.

A precondition such as:

```
Precondition
```

```
Assertion1
```

```
If( Assertion2 )
```

```
Assertion3
```

```
Assertion4
```

is equivalent to:

```
Precondition
```

```
Assertion1  && If( Assertion2 ) { Assertion3  && Assertion4 }
```

Likewise, an action clause such as:

```
Action
```

```
Action1
```

```
Action2
```

```
Modify
```

```
Variable1
```

```
Variable2
```

is equivalent to

```
Action
```

```
Action1; Action2; Modify { Variable1, Variable2 }
```

Annex B (normative)

Basic types and constants

B.1 Introduction

The data types that play a role in the M3W API can be subdivided into three categories:

- 1) Data types that are specific to an interface suite. These data types are defined in the API specification of the interface suite.
- 2) Data types that pertain to a group of functionally related interface suites, such as “video types” and “audio types”. These data types are specified in separate API specifications for each functional group.
- 3) Data types that are common to all interface suites. These data types, referred to as basic types, are defined in this API specification.

In defining the data types we will not use C data types such as `int`, `long`, `float`, etc. since the interpretation of these types, and in particular their size, may be implementation-dependent. Instead we use the names below to indicate implementation-independent types; these names (enclosed by angle brackets) are self-explanatory:

- `<n-bit signed integer>` (n = 8, 16, 32)
- `<n-bit unsigned integer>` (n = 8, 16, 32)

B.2 Public Types & Constants

B.2.1 Int8

Signature

```
typedef <8-bit signed integer> Int8, *pInt8;
```

Qualifiers

None.

Description

8-bit signed integer values.

B.2.2 Int16

Signature

```
typedef <16-bit signed integer> Int16, *pInt16;
```

Qualifiers

None.

Description

16-bit signed integer values.

B.2.3 Int32

Signature

```
typedef <32-bit signed integer> Int32, *pInt32;
```

Qualifiers

None.

Description

32-bit signed integer values.

B.2.4 UInt8

Signature

```
typedef <8-bit unsigned integer> UInt8, *pUInt8;
```

Qualifiers

None.

Description

8-bit unsigned integer values.

B.2.5 UInt16

Signature

```
typedef <16-bit unsigned integer> UInt16, *pUInt16;
```

Qualifiers

None.

Description

16-bit unsigned integer values.

B.2.6 UInt32

Signature

```
typedef <32-bit unsigned integer> UInt32, *pUInt32;
```

Qualifiers

None.

Description

32-bit unsigned integer values.

B.2.7 Double**Signature**

```
typedef <64-bit double precision floating point> Double, *Double
```

Qualifiers

None

Description

Compliant with IEEE standard:

The IEEE double precision floating point standard representation requires a 64 bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the exponent bits, 'E', and the final 52 bits are the fraction 'F'

B.2.8 Float**Signature**

```
typedef <32-bit single precision floating point> Float, *Float
```

Qualifiers

None

Description

Compliant with IEEE standard:

The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F'.

B.2.9 Char**Signature**

```
typedef <8-bit character> Char, *Char
```

Qualifiers

None

Description

8 bit character

B.2.10 Void

Signature

```
typedef void Void, *pVoid;
```

Qualifiers

None.

Description

The `Void` type has no values: it is used to indicate the absence of a return value or parameter list in functions. The `pVoid` type represents untyped pointer values.

B.2.11 Bool

Signature

```
typedef <32-bit unsigned integer> Bool, *pBool;
```

Qualifiers

— sub-type

Description

The Boolean values (`True` and `False`).

Constraints

For each `Bool b`

```
b == 0 || b == 1
```

B.2.12 Boolean Values

Signature

```
const Bool True = 1;
```

```
const Bool False = 0;
```

Qualifiers

None.

Description

These constants define the values contained in the data type `Bool`.

Constants**Table B.1**

Name	Description
True	The value representing the “true” condition.
False	The value representing the “false” condition.

B.2.13 String**Signature**

```
typedef <8-bit signed integer> *String, **pString;
```

Qualifiers

None.

Description

Null-terminated strings of 8-bit characters.

B.2.14 UUID**Signature**

```
typedef <128-bit unique identifier> UUID, *pUUID;
```

Qualifiers

None.

Description

128-bit unique identifier, compliant with the COM specification [13].

B.2.15 pIUnknown**Signature**

```
typedef <interface instance> *pIUnknown;
```

Qualifiers

None

Description

Interface reference