INTERNATIONAL STANDARD

ISO/IEC 23004-8

First edition
2009-04-01

Information technology — Multimedia Middleware —

Part 8: Reference software

Technologies de l'information — Intergiciel multimédia —

Partie 8: Logiciel de référence

Reference number
ISO/IEC 23004-8:2009(E)

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

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 23004-8 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

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

— *Part 1: Architecture*

— *Part 2: Multimedia application programming interface API*

— *Part 3: Component model*

— *Part 4: Resource and quality management*

— *Part 5: Component download*

— *Part 6: Fault management*

— *Part 7: System integrity management*

— *Part 8: Reference software*

Introduction

ISO/IEC JTC 1/ SC 29 has produced many important International Standards (for example MPEG-1, MPEG-2, MPEG-4, MPEG-7, and MPEG-21). One of the next steps in this process is the standardization of an Application Programming Interface (API) for Multimedia Middleware (M3W) allowing application software to execute multimedia functions with a minimum knowledge of the inner workings of the multimedia middleware as well as to support a structured way of updating, upgrading and/or extending the multimedia middleware.

Information technology — Multimedia Middleware —

Part 8:
Reference software

1 Scope

This part of ISO/IEC 23004 explains the organization of the reference software for ISO/IEC 23004– 1 to 7 (Multimedia Middleware). The electronic attachment to this part of ISO/IEC 23004 provides the source code of the actual software.

2 Normative references

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

ISO/IEC 23004-2, *Information technology — Multimedia Middleware — Part 2: Multimedia application programming interface (API)*

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

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

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

3 Overview of reference software

This is an informative clause. The reference software is organized into directories according to the different parts of ISO/IEC 23004. These directories are:

— **1_Architecture:** This directory is rather empty. The architecture is reflected by the implementation of the other parts.

— **2_Multimedia-API:** This directory contains the reference implementation of Audio and Video, Governance and IPMP logical components. The Audio and Video logical components are based on UHAPI4Linux implementation.

— **3_ComponentModel:** This directory contains the implementation of the core framework, services for remote method invocation (REMI) and services that allow instantiation of services based on a logical component id (Service Manager). The core framework also contains tools that aid in the development of M3W Components (IDL compiler).

— **4_ResourceManagement-Framework:** This directory contains the implementation of the resource management framework. This framework can be used to optimize the Quality of Service perceived by the user in a situation where resources are constrained and often not enough to run all applications and services at the highest quality level.

— **5_ComponentDownload-Framework:** This directory contains the implementation of the download framework. A framework that enables a large number of scenarios of controlled download / upload of components

— **6_FaultManagement-Framework:** This directory contains the implementation of the fault management framework enabling transparent addition of fault tolerance techniques to your software. The implementation consists of policies for intercepting the creation of services and tools for generating wrappers (middleman) that contain the fault tolerance techniques.

— **7_IntegrityManagement-Framework:** This directory contains the implementation of the integrity management framework targeted at maintaining and restoring in consistent software configuration on a device in the period that a device is owned and used by a consumer.

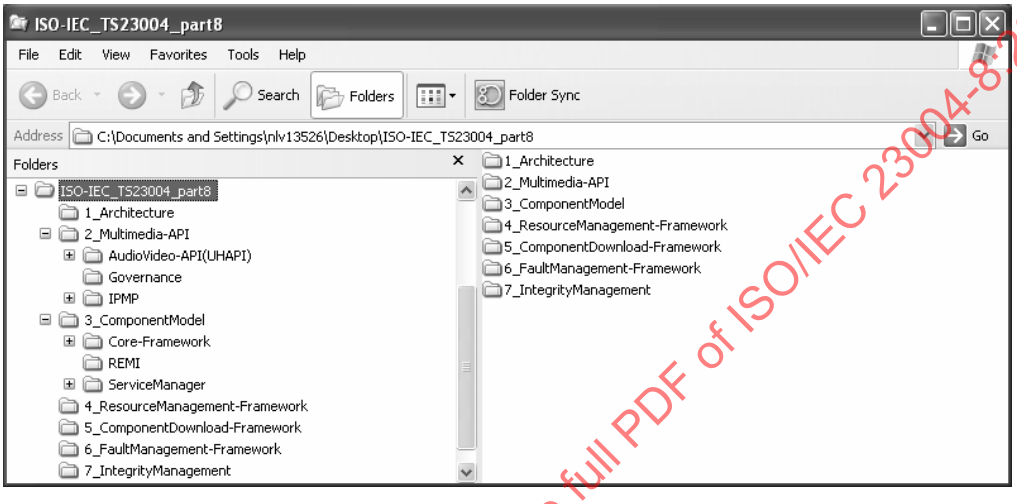

Figure 1 — Organization of the reference software in directories

4 Multimedia API

4.1 Introduction

This is an informative clause. This clause gives a brief explanation on the parts that together form the reference implementation for the Multimedia API of M3W.

4.2 Audio Video

4.2.1 General

The directory contains the current release of the reference implementation for clauses 7(general interfaces), 8 (audio interfaces) and 9 (video interfaces) of ISO/IEC-23004-2 (clause numbers are according to the FDIS version).

The implementation is based on the UHAPI4Linux implementation that can be found on Sourceforge (www.sourceforge.net).

4.2.2 Building

For building instructions please look at:

— `2_Multimedia-API\AudioVideo-API(UHAPI)\HOWTO`

4.2.3 Functionality

The reference implementation enables you to build a simple television application with features like changing channels, adjust volume, adjust contrast, adjust contrast, adjust saturation, image overlay, etc.

Figure 2 — Screenshot of Television Application build using reference implementation of Audio Video logical components

4.3 Governance

4.3.1 Introduction

The function of Security Manager is to provide secure interaction between M3W Service and external entities. This may include the interaction to acquire Service through download mechanism and invoke Service from remote peer. Security Manager also provides interface for governance/permission checking for accessing M3W Service and/or platform's resources.

4.3.2 Implementation

Currently the implementation consists of one executable component with a SecurityManager service that implements the ISecurityManager interface suite (logical component) and an example client application that uses this service.

Service and application are build in the following way:

- autoreconf –is
- ./configure –prefix=$HOME/local
- make
- make install

Please remember to register the component and its service after installation but before execution of the client application. This is done by the following command:

- rcregtool < regscript

The registration script "regscript" is generated during the build process.

4.4 IPMP

4.4.1 Introduction

ISO/IEC 23004-2 M3W multimedia API has been developed as a reference software. This clause describes the reference software that provides the M3W middleware interface for MPEG-2 IPMP [3], MPEG-4 IPMPX [4], and MPEG-21 IPMP [5] services. It will refer to IPMP interface for supporting IPMP service of a device part of M3W.

The implementation is M3W complaint except for:

— one operation name in the `rcIComponent (getServiceFactory())` and the interface name `rcIServiceFactory`. The current release still uses the legacy names `getServiceManager` and `rcIServiceManager`.

— The usage of a service specific interface instead of the `rcIServiceGeneric` interface

The IPMP interface suite is listed as following Table 1.

Table 1 — IPMP interface suite list

IPMP Interface Suites		
Name	Path	Functional Description
Trust Management		
Key Management	~/IPMP/component /trustmgmt/keymgmt	Key management component is essential for IPMP functions such as encryption, decryption, watermarking, and authentication and is used to process encrypted multimedia data in accordance with getting and generating key information.
Signature Management	~/IPMP/component /trustmgmt/signmgmt	Signature management component is important to represent immunity of IPMP properties. A digital signature means a value generated from the application of a private key to a message via a cryptographic algorithm such that it has the properties of integrity, message authentication and/or signer authentication.
License Management	~/IPMP/component /trustmgmt/licensemgmt	In DRM system, usage rule and copyright are written in a license with key information. License management component is a representation of right expression for digital contents.
Certificate Management	~/IPMP/component /trustmgmt/certmgmt	Certificate management component can be used in managing key and related information.
Domain Management	~/IPMP//component /trustmgmt/domainmgr	Domains management component are supporting groups of devices or users to share some common properties. These are defined to manipulate domain elements such as domain information, device information and so on.

Tool Interfaces		
General Tool Processing	~/IPMP/component /tool/toolproc	According to tool processing component, the required IPMP tools can be selected and initialized. After that, the multimedia data are processed by tool processing component and the selected tool dose its duties based on IPMP messages.
IPMP Tool Functions	~/IPMP/component /tool/toolfunc	Any kind of functions can be supported in a device, if the functions are related to IPMP tool functions component. However, IPMP tool functions component defines a tool function of general functionalities for DRM or IPMP services such as watermark and authentication functions.
Tool Update	~/IPMP/component /tool/toolupdate	Tool update component should support updating and installing IPMP tool and upgrading the functionality of the tool.
Tool Communications	~/IPMP/component /tool/toolcom	Communications between different tools or IPMP devices should be supported by tool communication component.

Table 1 contains path name and functional description with implemented component name. And all components are implemented by C/C++ language.

We have classified components with *trust management* interfaces and *tool* interfaces. The trust management interfaces are to support tool's functionality by accessing secured information. The tool interfaces are to perform a variety of IPMP functions based on IPMP information. Also, components are developed in IDL (Interface Description Language) and IDL type's components have been compiled by M3W IDL compiler. After that, IDL type's components will be C/C++ type components. For more details of IDL compiler usage, we recommend referring to ISO/IEC 23004-3.

The RIDL type of component needs common type definition for IPMP interface suite. Therefore, we have defined types in a file "ipmp_types.ridl." When "ipmp_types.ridl" is compiled by IDL compiler, we can get two files; "ipmp_types_common.c","ipmp_types_common.h." These two files are required whenever our proposed component is registered in M3W runtime environment.

4.4.2 Example of IPMP Interface Suites

We have implement examples for verifying components for IPMP interface suite whether it is right to operate on M3W. A set of IPMP interface suite example is listed in Table 2. Note that the following examples are just calling interface functions of a component on M3W environment and their components are not actual operation for IPMP service.

Table 2 — Example of IPMP interface suites

Example		Path
1	IPMP processing	~/IPMP/example/IPMPprocessing
2	License management	~/IPMP/example/Licensemangement
3	Certificate management	~/IPMP/example/Certmanagement
4	Authenticating using a certificate	~/IPMP/example/Authentication
5	Add new domain	~/IPMP/example/AddNewDomain
6	Revoke domain	~/IPMP/example/RevokeDomain
7	Update domain	~/IPMP/example/UpdateDomain
8	IPMP Processing extension	~/IPMP/example/IPMPprocessingExt

Also, we can find out execution files for examples in the above path when archived file is installed in M3W environment.

The following is the description for IPMP interface suites example. Although examples do not have the functional process for IPMP service, they effectively show how to call an interface function of IPMP service component.

4.4.2.1 IPMP processing

IPMP terminal has all tools and do not need to find the missing tools. IPMP tool manager of IPMP terminal requests M3W's service manager to get the service instance of tools. And IPMP tool manager requests each tools to start `operateTool()`. This could be done by using `Start()` method in `RcIService` interface that is implemented by the IPMP tool service.

4.4.2.2 License management

License management component is how to check and get a valid license through license management component. When a tool requires license information, license management component should get and parse a valid license. After that it should return a result into the tool.

4.4.2.3 Certificate management

To certificate, we need operations such as issuing, updating, revoking for certificating a user with a certificate management component of M3W environment.

4.4.2.4 Authenticating using a certificate

When a device or a component has its certificate, it should be ready for authentication. If a device *A* wants to authenticate a device *B*, the device *A* requests the device *B*'s certificate and should validate it.

4.4.2.5 Add new domain

Domain Manager (DM) requests authentication of domain information such as administrator to Authenticator. Once Authenticator receives the required information from DM, Authenticator verifies the user which will be the domain administrator. If the information is valid, DM creates new domain and returns the result to Initiator.

4.4.2.6 Revoke domain

To remove information of a domain, an administrator requests DM to remove the domain. Once DM receives domain removal message from a user device, the message is transmitted to Authenticator for verifying. And then DM revokes the domain information and it returns the result to a user device.

4.4.2.7 Update domain

To update a domain, a user device requests updating domain to DM. When the DM receives the domain update message from a user device, the message is transferred to Authenticator for verification. If the information is authenticated, DM updates domain and returns the result to Initiator.

4.4.2.8 IPMP processing extension

The IPMP information of MPEG-2 IPMP, MPEG-4 IPMPX is extracted and then IPMP tool is executed. Specifically, according to the IPMP information, the proper IPMP tools are retrieved and then executed. If there is no proper tool, a tool manager should get missing tools from remote URI in the IPMP information. During the processing of MPEG-2/4, whenever and wherever IPMP processing is required, each tool performs its own IPMP functions.

5 Component model

5.1 Core framework

5.1.1 General

This directory contains the current release of the reference implementation for ISO/IEC-23004-3. This release does not contain:

— services needed for remote method invocation (as specified in subclause 6.2)

— servicemanger (as specified in subclause 6.1.2)

The references for these parts can be found in the other subdirectories of 3_ComponentModel.

5.1.2 Building

For information on how to build and install the software please look at the following file:

— 3_ComponentModel\Core-Framework\README

5.1.3 Functionality

This directory contains the software needed for instantiation of services. This library is the runtime environment. The directory contains also tools that aid in the development with components that are compliant with the M3W component model (IDL compiler). The M3W Component model specifies:

— Unit of Trading (*M3W Component*)

— Unit of Loading (Executable Component)

— Unit of Instantiation (*Service*)

— Standard interfaces

  — Navigation between interfaces (*rclUnknown, rclServiceGeneric*)

— Binding (rclServiceGeneric)

— Access to public attributes (*rclServiceGeneric*)

5.2 REMI

5.2.1 Introduction

M3W manages the execution of services deployed in components. These services are used by means of late binding, i.e. a service and a requester are bound only when the latter actually needs to execute the former.

By default configuration this mechanism uses only local components, i.e. those deployed on the device and previously registered to the local runtime process.

Inside the M3W two logical components have been included as optional. The benefit is to have the possibility to issue request to service instance that are located in other M3W Systems by means of network messages. This opens to sharing of functionalities, to distributed computing for fault tolerance/load balancing purposes.

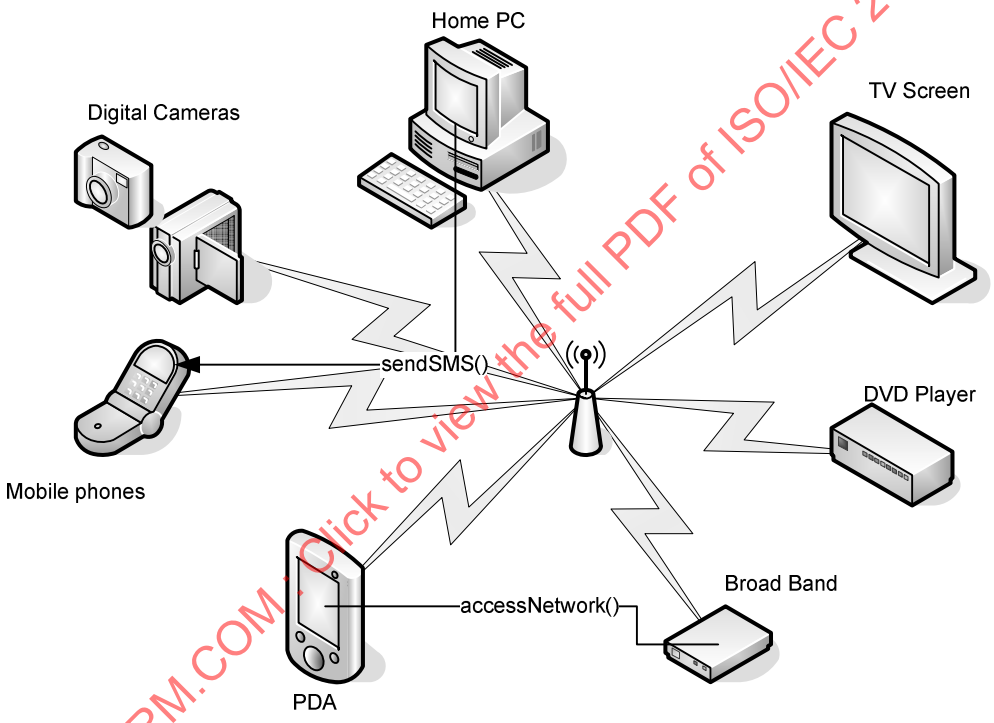

Figure 3 — Sharing of M3W functionalities among devices at home

New generation of consumer electronic applications can be realized in order to exploit/manage the functionality that are discovered inside the distributed environment (e.g. home wi-fi network).

The high-level exploitation is based on a low-level control of the services which are exposed by any M3W System in a distributed environment. A logical view of applications and functionalities interacting with other functionalities not necessarily located in the local system is depicted in the following figure.

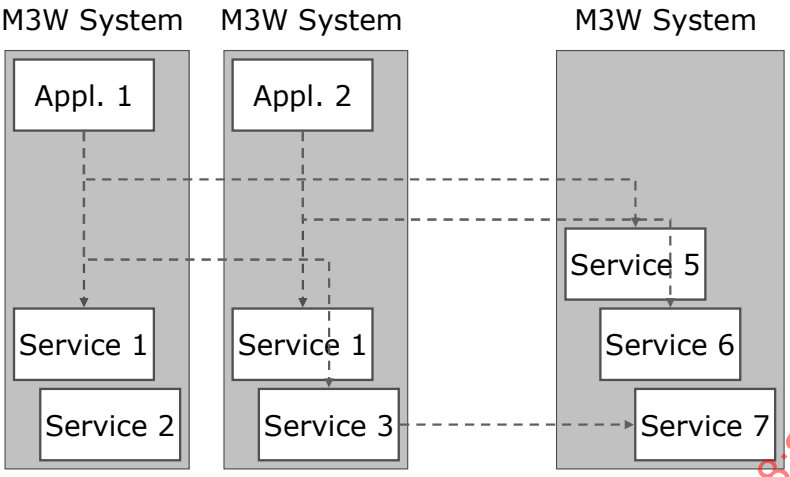

Figure 4 — Use of service by client code: local ore remote instances

In order to achieve this objective, a stack of communication layers is needed. The logical components which have to be at disposal to allow enabling the transparent execution of remote component services are resumed in the following:

— a remote requester (REMI-R) that allows client code to issue method execution requests to instances (new or existing) to the other M3W Systems connected to the network.

— a remote provider (REMI-P) that is able to accept and to handle method requests by redirecting them to the proper “active” instances. It is also capable to respond about service availability, and with services/interfaces metadata.

— server and client for Remote Procedure Call (RPC).

Please note, that since these logical components are not mandatory, several device configurations for Remote Capabilities are allowed (e.g. “only-client” devices can be configured by providing an implementation for the REMI-R and the RPC client logical components). In order to allow a “complete” solution for Remote Capabilities exploitation both REMI-R and REMI-P, with the corresponding RPC components, are recommended. The REMI-P is needed when client code needs to use a remote service passing as argument an interface reference to a local instance; in this case the REMI-P has to be configured in order to handle the method requests addressing the passed reference.

5.2.2 Implementation

All the software is build in the same way, using the following commands:

- autoreconf –is
- ./configure –prefix=$HOME/local
- make
- make install

5.3 Service Manager

5.3.1 General

Files: The source code can be found in the directory 3_ComponentModel\ServiceManager
Description
C source code for the implementation of M3W Service Manager. The code implements • APIs for Service Manager • Parsers and interpreters for Logical Component metadata • Parser and interpreter for Service metadata
Input
Logical Component and Service metadata
Output
None
Programming Language(s)
C
Platform(s)
Linux
Dependencies
Libxml2 library for Linux

The implementation is M3W complaint except for:

— one operation name in the `rcIComponent (getServiceFactory())` and the interface name `rcIServiceFactory`. The current release still uses the legacy names `getServiceManager` and `rcIServiceManager`.

— The usage of a service specific interface instead of the `rcIServiceGeneric` interface

5.3.2 Dependencies

To build the M3W Service Manager, we must first install the core framework. The steps of the installation can refer to the documents along with core framework source code. The installation should be executed on the Linux platform with g++, make, libc6-dev, autoconf, automake, libtool, bison, flex, sqlite, and libxml2.

We assume that the position of the core framework installation is “/usr/local/”. So we have rccommon, rre, and ridl library installed on the system.

5.3.3 Building the software

After installing the core framework on the operating system, we can begin to install the Service Manager to the system. We implement Service Manager as a special service running on Roboarch framework which should be known by all client applications. The building process is as following:

1) Make sure there is libxml2 installed in the system.
2) Copy the iSMControl.ridl, iSMClient.ridl, sServiceManager.ridl, and cServiceManager.ridl from the source code package to a building folder and run the command:
```
ridl –skel *.ridl
```
3) Copy the ServiceManager_SServiceManager_Impl.c/h Metadata.c/h, acinclude.m4, Makefile.am, configure.ac, regscript.in, AUTHORS, ChangeLog, COPYING, NEWS, and README to the building folder. Then run the command:
```
autoreconf –is
```
4) Then we can begin to compile and build the Service Manager by running the following commands:
```
./configure

make

make install
```
5) Then Service Manager is installed in the position /usr/local as a library libServiceManager.so

5.3.4 Using the reference software

5.3.4.1 The Service Manager interfaces

There are two interfaces are defined in the ISO standard for Service Manager.

1) rcISMClient
Client can access service instance through this interface. Client only needs to know which logical component contains the functionality it requires, and it will get the service instance implements the functionality by using this interface.

There are two operations in this interface, they are related to query service and get service instances.

The operations are as follows:

```
rcResult isServiceAvailable(in pUUID lcID, out Bool retValue);
```
We can use this operation to check the availability of implementation for a specific logical component. If there is service implements the logical component with UUID lcID, this operation will return true.

```
rcResult getInstanceForLogicalComponent(in pUUID lcID, out prcIService *retValue);
```
This operation will return the service reference of the service corresponding to the required logical component. Service Manager will search in its metadata collections to find the service implements the logical components, and then get the service reference from RRE and return it.

2) rcISMControl
This interface provides optional operations to manage the metadata collections. Service Manager should record all metadata files and parse the files to form a dependency relationship tree for all logical components and services.

There are four operations defined in this interface, they are used to register and unregister the logical component and service metadata files.

```
rcResult registerLogicalComponentMetadata (in String path, out Bool retValue);
rcResult unregisterLogicalComponentMetadata (in pUUID lcID, out Bool retValue);
```

The two operations will add and remove the logical component metadata to the Service Manager.

```
rcResult registerServiceMetadata (in String path, out Bool retValue);
rcResult unregisterServiceMetadata (in pUUID srvID, out Bool retValue);
```

The two operations will add and remove the service metadata to the Service Manager.

All the metadata files records are stored in two files: /usr/local/metadata/components and /usr/local/metadata/services. When Service Manager is running, it will load all the recorded metadata files and parse them. The parsed logical component/service structures will be stored in memory for later query and other operations.

We use the local files to store the records of metadata files because the shared library implementation in Linux can't guarantee the singleton of Service Manager. This means there will be one instance of Service Manager and RRE for one client applications. So we can't keep the metadata records in memory but have to write them into a local file. This file will be shared by all instances of Service Manager.

5.3.4.2 Installation of the components and clients

With Service Manager, the process of a client application can be like this:

1) If the Service Manager begins with empty service registered, there should be one application (client or middleware administrator's application) that registers the services and logical components. This step only need run once. If you register the same metadata file multiple times, the metadata parser will ignore the duplicated ones.

2) Once the services and logical components are registered, client applications can begin to run and use Service Manager. Client applications need to know only logical component, which means it knows which logical component can provide the functionalities it needs. But to use the required functionalities, client application must inquiry Service Manager to get the service instance that provides the logical component functionalities.

3) Through Service Manager, client applications can get the service instance and begin to use the provided functionality.

To install a logical component to the M3W system, we can follow the steps introduced by Roboarch to install the logical component/service/interface suite. After installation, we should write the logical component and service metadata and register them. The example of Logical Component metadata can be as the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<M3WLC xmlns="http://mpeg-m3w/MM_APIs" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://mpeg-m3w/MM_APIs D:\MPEG\Schemas\M3WLogicalComponent.xsd" version="1.0">
    <LogicalComponents>
            <ParamClassTypes>
                <ParamClassType name="uhErrorCode_t">
                    <Description>Some Description on parameter type </Description>
        </ParamClassType>
                … There are many parameters type declarations
            </ParamClassTypes>
            <LogicalComponent id=”UUID of logical component"  name="component      name">
        <Description>Description on the component</Description>
                <Role id="UUID of service" name="service name">
                    <Interface id="UUID of interface" name="interface name">
                        <Method name=" " output=" ">
                        <Parameter name=" " type=" " />
                            … There may be several parameters in one operation
                        </Method>
                            … In one interface, there may be several operations
                        </Interface>
                </Role>
            </LogicalComponent>
            … Many logical components can coexist in one metadata file
        </LogicalComponents>
</M3WLC>
```

The example of Service metadata is as the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Component xmlns="http://mccb.icu.ac.kr/m3w" xmlns:UED="urn:mpeg:mpeg21:2003:01-DIA-NS"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://mccb.icu.ac.kr/m3w
   D:\MPEG\Schemas\M3WService.xsd">

   <ComponentID>The UUID of component </ComponentID>
   <ComponentTitle>The component title</ComponentTitle>
   <ComponentDescription>Description of the component </ComponentDescription>
   <Creator>
     <CreatorName>The component creator</CreatorName>
  </Creator>
   <AvailableInterfaces>
     <Interface>
         <InterfaceID>UUID for Interface</InterfaceID>
         <InterfaceName>Name of Interface</InterfaceName>
         <InterfaceImplementors>
             <ServiceID>UUID for service</ServiceID>
             …  There may be many services relates to one interface
         </InterfaceImplementors>
```

```
    </Interface>
    …  There may be many interfaces
  </AvailableInterfaces>
  <Services>
    <Service>
        <ServiceID>UUID for service</ServiceID>
        <ServiceInvocationType>in-process</ServiceInvocationType>
        <RequiredInterfaces>
            <InterfaceID>UUID for required interface</InterfaceID>
            … One service may depends on other services
        </RequiredInterfaces>
    </Service>
    … One component may have several services
  </Services>
</Component>
```

To register the logical component metadata file, we can add the file path to the "/usr/local/metadata/Components", to register the service metadata file, we can add the file path to "/usr/local/metadata/Services". We should make sure there are no duplicate paths in two files.

Also we can register metadata files dynamically by using register functions provided by Service Manager Control interface before using the service functionality.

5.3.4.3 Application example

We implement two example applications and two example logical components to show the functions of Service Manager. The two example components are Hello1 and Hello2; they both provide a simple operation is to display a hello world message. We can install the two components follow the steps of installing a component to system in Roboarch.

The client application App_Reg is responsible for register the Hello1 and Hello2 metadata to Service Manager. This application only needs to run once. The metadata for Hello1 and Hello2 Logical Component is as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<M3WLC xmlns="http://mpeg-m3w/MM_APIs"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://mpeg-m3w/MM_APIs D:\MPEG\Schemas\M3WLogicalComponent.xsd"
version="1.0">
   <LogicalComponents>
      <LogicalComponent id="89464348-734a-488c-ab7e-80682d1bf76f" name="Hello1">
         <Description>Example Logical Component 1 to show M3W Service Manager functions
         </Description>
         <Role id="f5a3fa31-0dd7-4c55-b38c-2fbb82664a74" name="SHello1">
            <Interface id="d62d7e91-9474-4b84-b05f-8fe9b413a86f" name="IHello1">
               <Method name="hello1" output="void"/>
               </Interface>
         </Role>
      </LogicalComponent>
      <LogicalComponent id="98464348-734a-488c-ab7e-80682d1bf76f" name="Hello2">
         <Description>Example Logical Component 2 to show M3W Service Manager functions
         </Description>
         <Role id="e4a3fa31-0dd7-4c55-b38c-2fbb82664a74" name="SHello2">
            <Interface id="e52d7e91-9474-4b84-b05f-8fe9b413a86f" name="IHello2">
               <Method name="hello2" output="void"/>
            </Interface>
         </Role>
      </LogicalComponent>
   </LogicalComponents>
</M3WLC>
```

Functionalities of the above Logical Components are provided by Services described in the two following metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<Component xmlns="http://mccb.icu.ac.kr/m3w"
xmlns:UED="urn:mpeg:mpeg21:2003:01-DIA-NS"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://mccb.icu.ac.kr/m3w D:\MPEG\Schemas\M3WService.xsd">
   <ComponentID>89464348-734a-488c-ab7e-80682d1bf76f</ComponentID>
   <ComponentTitle>Hello1</ComponentTitle>
   <ComponentDescription>Example component 1. This is just example</ComponentDescription>
   <Creator>
      <CreatorName>MCCB Lab</CreatorName>
   </Creator>
   <AvailableInterfaces>
      <Interface>
         <InterfaceID>d62d7e91-9474-4b84-b05f-8fe9b413a86f</InterfaceID>
         <InterfaceName>IHello1</InterfaceName>
         <InterfaceImplementors>
            <ServiceID>f5a3fa31-0dd7-4c55-b38c-2fbb82664a74</ServiceID>
         </InterfaceImplementors>
      </Interface>
   </AvailableInterfaces>
   <Services>
      <Service>
         <ServiceID>f5a3fa31-0dd7-4c55-b38c-2fbb82664a74</ServiceID>
         <ServiceInvocationType>in-process</ServiceInvocationType>
      </Service>
   </Services>
</Component>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Component xmlns="http://mccb.icu.ac.kr/m3w" xmlns:UED="urn:mpeg:mpeg21:2003:01-DIA-NS"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://mccb.icu.ac.kr/m3w D:\MPEG\Schemas\M3WService.xsd">
   <ComponentID>98464348-734a-488c-ab7e-80682d1bf76f</ComponentID>
   <ComponentTitle>Hello2</ComponentTitle>
   <ComponentDescription>Example component 2. This is just example</ComponentDescription>
   <Creator>
      <CreatorName>MCCB Lab</CreatorName>
   </Creator>
   <AvailableInterfaces>
      <Interface>
         <InterfaceID>e52d7e91-9474-4b84-b05f-8fe9b413a86f</InterfaceID>
         <InterfaceName>IHello2</InterfaceName>
         <InterfaceImplementors>
            <ServiceID>e4a3fa31-0dd7-4c55-b38c-2fbb82664a74</ServiceID>
         </InterfaceImplementors>
      </Interface>
   </AvailableInterfaces>
   <Services>
      <Service>
         <ServiceID>e4a3fa31-0dd7-4c55-b38c-2fbb82664a74</ServiceID>
         <ServiceInvocationType>in-process</ServiceInvocationType>
      </Service>
   </Services>
</Component>
```

When client application Run_App runs, it will first request instance of Service that implement functionality of Logical Component with the component id: "89464348-734a-488c-ab7e-80682d1bf76f"; then after that, it will request instance of Service that implement functionality of Logical Component with Component id: "98464348-734a-488c-ab7e-80682d1bf76f"; The sequence diagram of Run_Register, Run_App, and Service Manager is as the following:

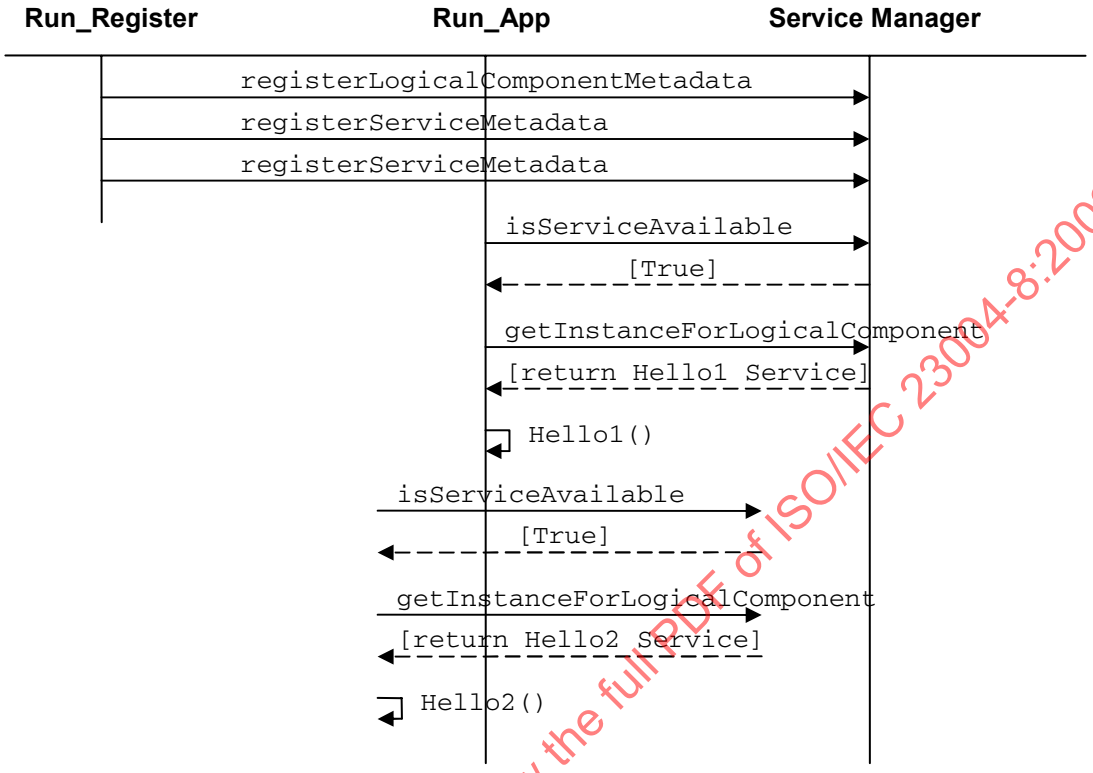

Figure 5 — Sequence diagram for the example application

6 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. 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.

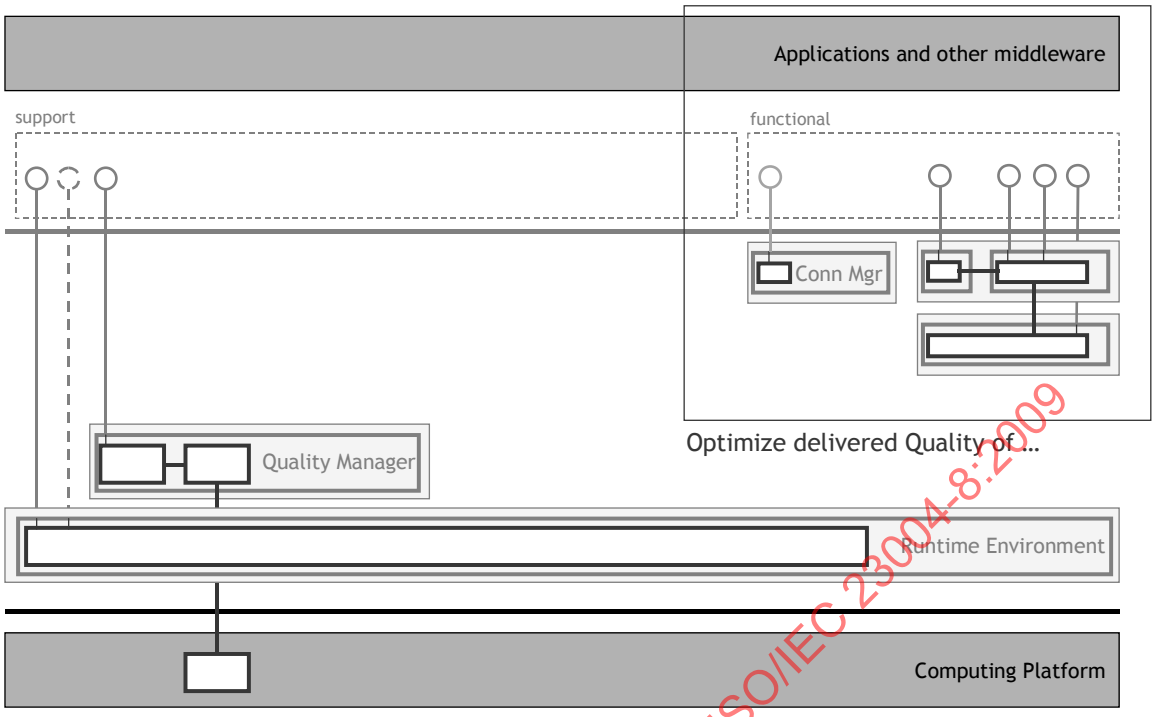

Figure 6 — M3W Resource Management Framework

The resource management reference code can be found in the following directory: 4_ResourceManagement-Framework.

All services and applications are build in the following way:

- autoreconf –is
- ./configure –prefix=$HOME/local
- make
- make install

7 Component download

7.1 Overview

The reference software for ISO/IEC 23004-5 consists of 3 parts:

— **Library:** This contains an implementation of the functionality of all the roles used in the download framework (target, initiator, locator, repository, decider). The functionality provided by this library can be used in implementing for example a target service, initiator service, or repository application.

— **Comps:** This contains a component with a target service and initiator service. Which represent the two roles that are most likely to be deployed on a consumer device.

— **Apps:** This contains example code on how to use the services

7.2 Building

Building sequence:

— library

— comps

— apps

Building instructions (the same for all modules):

— autoreconf –is

— configure –prefix=<path>

— ./configure

— make

— make install

The components need to be registered at the runtime environment after they have been installed but before the application is started. For this purpose a registration script is provided in the comps directory (rcregtool < regscript). Start the Runtime environment before executing this script.

7.3 Functionality

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.

8 Fault management

8.1 Overview

The reference software for Fault management consists of the following modules

— **Compiler (compiler):** This module contains the FXL & XML compiler to make middlemans. The FXL stands for fault management extension language. Based on a fault management extension description the compiler is able to generate a middleman for a specific service that contains / embeds a number of specified in FXL. The compiler is added for illustration. The code that is generated is not fully M3W compliant (95%). The generated code still needs to be modified manually in order to be fully M3W compliant.

— **Instantiation Policy (fmpo):** The runtime environment supports pluggable instantiation policies. These policies influence the instantiation of services. In this case fmpo enables the instantiation of a middleman wrapping a service when that specific service is requested.

— **Example Middlemans and Demo scenario (faultManagementDemo):** The possibilities for generating middlemans and fault tolerance scenarios is almost infinite. It is impossible to demonstrate them all. However an example on how to use the framework is added.

8.2 Compiler

8.2.1 General

The directory compiler contains the FXL & XML compiler. This compiler is added for illustration. The main idea behind the middleman is that they are generated and not written manually (otherwise you might introduce more faults than are resolved). The compiler is not fully M3W compliant, but the generated code can be made fully compliant manually (a recipe is provided in the form of a Powerpoint presentation and of course the example scenario).

8.2.2 Building

Please use the following build instructions:

— Using Visual Studio .Net (Windows Environment)

— Open the solution _RobocopModel.sln

— Set the RoboCopDriver (XML Compiler) as startup project

— Build the executable, it will be build in:
RobocopDriver/bin/Debug or Release

— Set the FxlDriver (FXL Compiler) as startup project

— Build the executable, it will be build in:
FxlDriver/bin/Debug or Release

8.2.3 Functionality

The compiler is able to generate the complete implementation of a Middleman that adds fault tolerance mechanisms to a service. The fault tolerance mechanisms are not specified in M3W because this is the area where different middleware vendors are able to distinguish themselves.

The compiler is able to generate code for a limited number of fault tolerance mechanisms based on a fault management extension language. It is not the intention that people are going to use the compiler as is, but merely to demonstrate the feasibility and to serve as an example. For that reason the compiler will not be updated in order to generate 100% compatible M3W code.

The compiler consist of two parts:

— XML Compiler for generation of skeleton code

— FXL Compiler for generation of code for the fault tolerance mechanisms

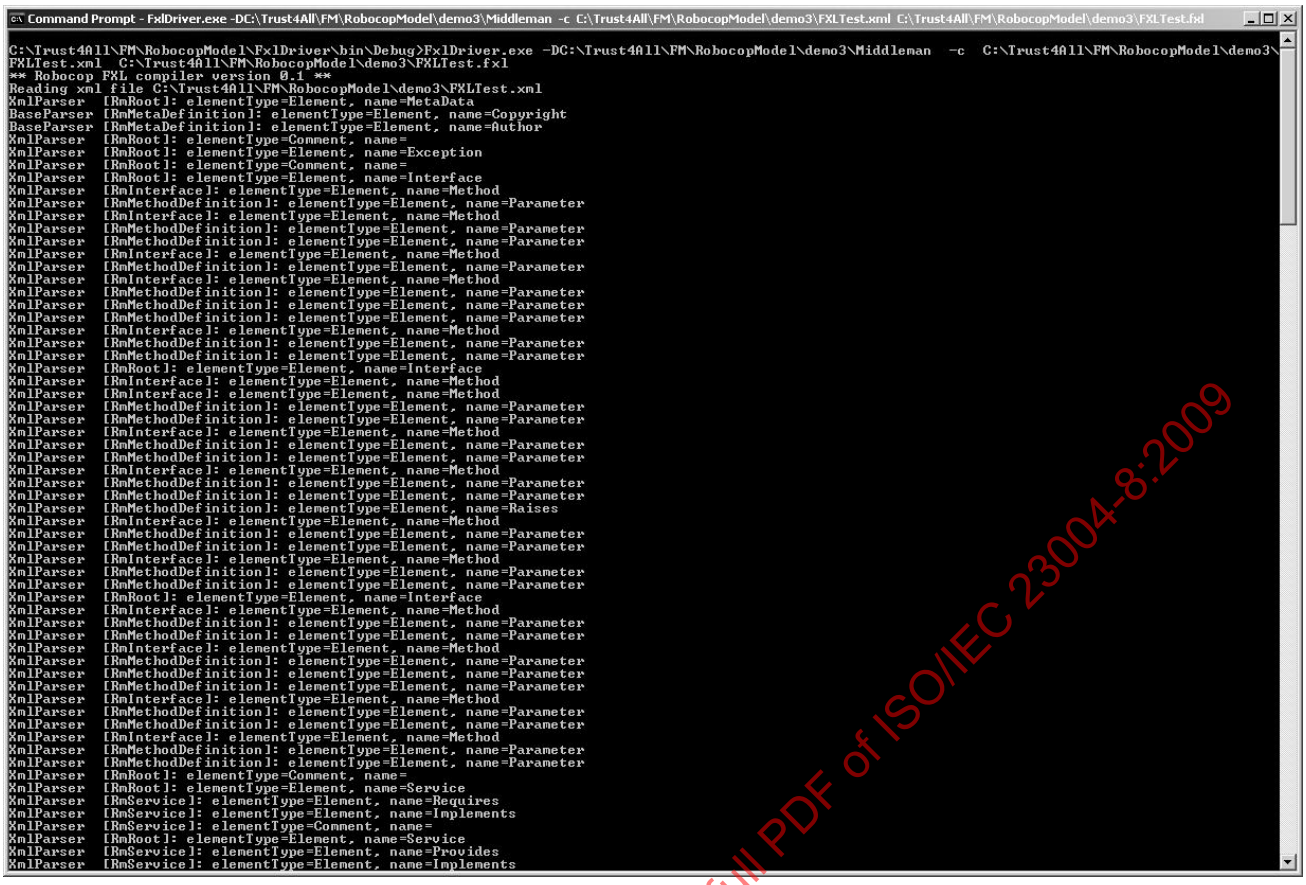

Figure 7 — XML Compiler in action

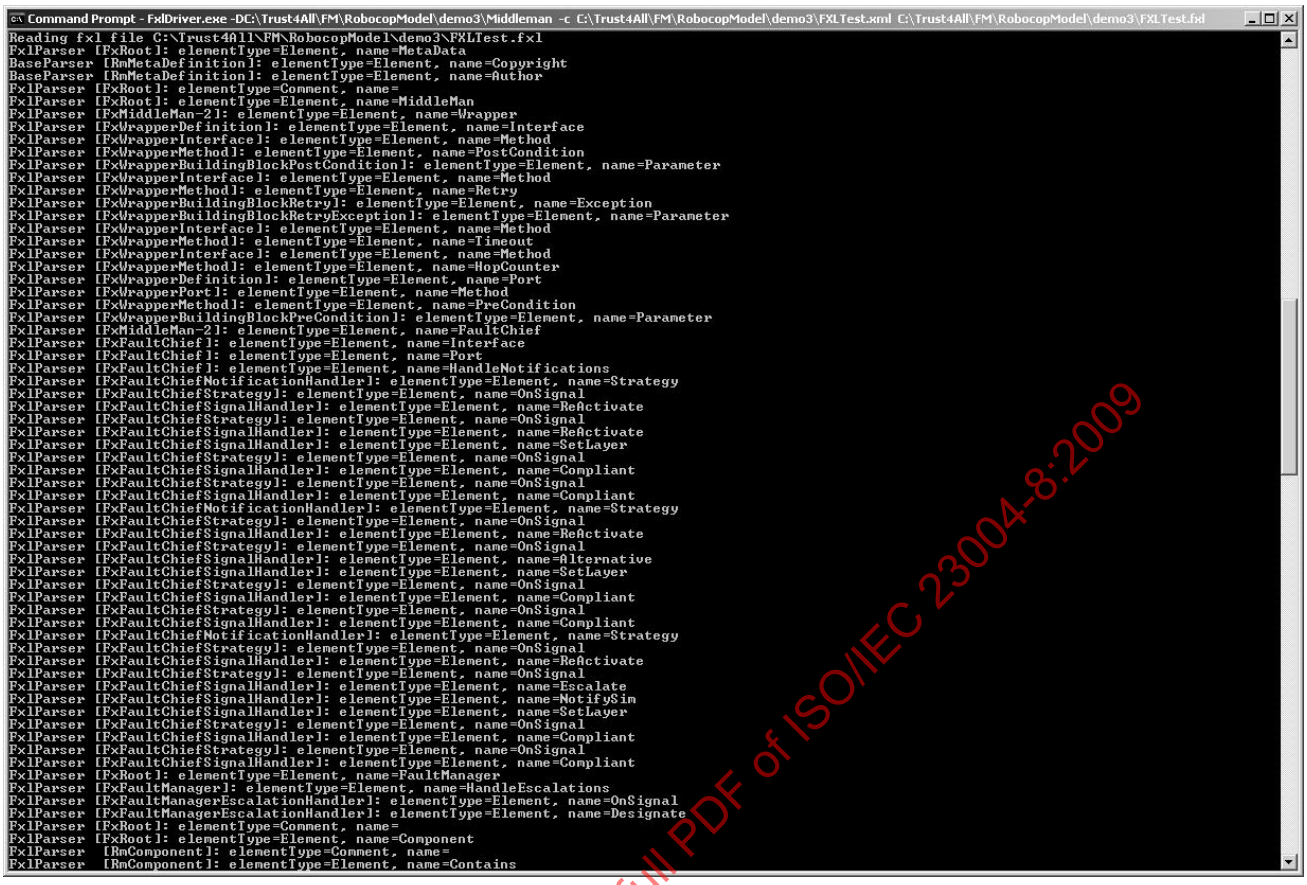

Figure 8 — FXL Compiler in action

8.3 Instantiation policy

8.3.1 General

The directory fmpo contains the (compile time) pluggable fault management instantiation policy.

8.3.2 Building

In order to build follow the following instruction

— First build the "roboarch" environment as seen before.

— cd <path to "fmpo">

— autoreconf -is

— ./configure "CFLAGS=<your favorite CFLAGS>" --prefix=<your install root>
In our case : ./configure --prefix=$HOME/local

— make

— make install

— cd <path to "roboarch">

— ./configure "CFLAGS=<your favorite CFLAGS>" --prefix=<your install root> --with-policy=Fmpo
In our case : ./configure --prefix=$HOME/local --with-policy=Fmpo

— make

— make install

8.3.3 Functionality

The Fault Management Instantiation policy enables the runtime environment to automatically wrap specific services when they are instantiated.

8.4 Example middleman and demo scenario

8.4.1 General

The directory faultManagementDemo contains a very simple example on how to add fault tolerance mechanisms to a service using the fault management framework specified in ISO/IEC 23004-6. The example is not intended to demonstrate the full strength of these techniques but more as a small tutorial that is not polluted by additional complexity.

8.4.2 Building

In order to build follow the following instructions:

— cd <path to "example_applications\faultManagementDemo\Component">

— autoreconf -is

— ./configure "CFLAGS=<your favorite CFLAGS>" --prefix=<your install root>
In our case : ./configure --prefix=$HOME/local

— make

— make install

— Start the rreX in a separate window by entering: rreX

— sh regscript

— Stop the rreX by pressing ctrl+c in the window where it is started.

— cd <path to "example_applications\faultManagementDemo\Middleman">

— autoreconf -is

— ./configure "CFLAGS=<your favorite CFLAGS>" --prefix=<your install root>
In our case : ./configure --prefix=$HOME/local

— make

— make install

— Start the rreX in a separate window by entering: rreX