



FD – Execution Engine

Content

1	INTRODUCTION	3
2	FUNCTION STRUCTURE	3
2.1	APPLICATION	3
2.2	COMPILER	4
2.3	RUNTIME	4
2.3.1	Event handling	5
2.4	COMPONENT INTERFACE	5
2.5	PLATFORM ACCESS	5
3	FUNCTION BEHAVIOR	5
3.1	STARTING AN APPLICATION EXECUTION	5
3.2	COMPILING AN APPLICATION	6
3.2.1	Relationships between compilation and execution	6
3.2.2	Compiling a source file into a Module	6
3.2.3	Operations	7
3.3	EXECUTION	7
3.3.1	Relationships between Application and Runtime	8
3.3.2	Starting the execution	8
3.3.3	Engine algorithms	8
3.3.4	Program state	9
3.3.5	Invoking of a Product	9
3.3.6	Executing an Operation	9
3.3.7	Unwinding the stacks	9
3.3.8	Event handling	10
3.3.9	Design principles of runtime classes	12
3.4	EXECUTING A VOICEXML SUBDIALOG	13
3.5	STABILITY PRINCIPLES	14
3.6	RUN-TIME BEHAVIOR	14
3.7	PLATFORM ACCESS	14
3.7.1	Object identities	14
3.7.2	Error Handling	14
3.8	PLATFORM ACCESS	15
3.9	RESOURCE LOCATING	15
3.9.1	Cache	15
3.9.2	HTTP resource locating	16
3.9.3	Sequence diagrams	16
4	APPENDIX A: EXAMPLES OF GRAPHS OF EXECUTABLES AFTER COMPILATION	18



Approved: Magnus Björkman		Mobeon Internal No: 3/FD-MAS0001 Uen	
Copyright Mobeon AB All rights reserved	Author: ERMKESE QDALO MPAZE QMIAN EJEFRID Title: FD – Execution Engine	Version: B Date: 2008-08-05	2/22

4.1	DESCRIPTION OF THE OPERATIONS USED IN THIS SECTION	18
4.2	EXAMPLE 1	18
4.3	EXAMPLE 2	19
5	APPENDIX B: 3PPS	21
5.1	THIRD-PARTY PRODUCTS AND FREWARE	21
6	REFERENCES	22
7	TERMINOLOGY	22

Figures

Logical packages	3
Event handling structure in EE with active CCXML and VXML documents	11
Event processing	12
Data not found in cache and fetched from server.	16
Data in cache has aged, but http server says it's valid.	17

Tables

Sample operation suffixes	7
---------------------------	---


Key execution engine facts explained

- ☞ What is compilation?
- ☞ How is operations executed?
- ☞ How are sequences of operations composed?
- ☞ Is an operation aware of adjacent operations?

History

Version	Date	Adjustments
A	2006-10-05	First revision (ERMKESE).
B	2008-08-05	Added VCP Resource Locating (3.9) (EJEFRID).

1 Introduction

This document describes the Execution Engine. Some parts are crucial to understanding the how the Execution Engine, or EE for short, works. These parts have the symbol  in the left margin.

2 Function structure

The Execution Engine consists of the following major parts:

- Application: A representation in compiled form of a collection of CCXML, and VoiceXML documents.
- Compiler. Compiles XML documents to an executable representation. There is one VoiceXML compiler and one CCXML compiler.
- Runtime: Executes an application.
- Component interface: Implements the external interfaces that defines the EE as a component.

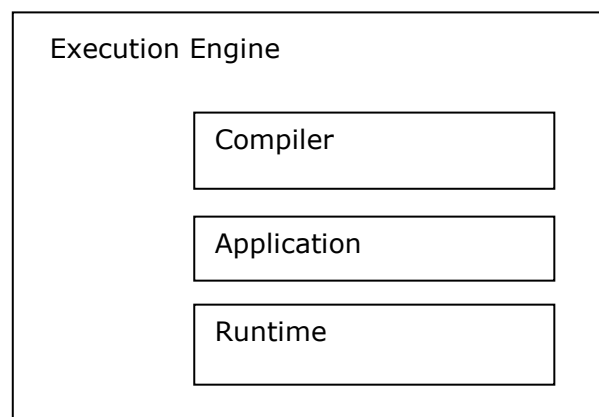


Figure 1, Logical packages

2.1 Application

This logical package contains various classes defining Applications.

The following are the primary entities that belong to the Application package:

- Application. An application is a compiled version of a CCXML/VoiceXML program. An application consists of a set of Module instances, one per source file.
- Module. A module contains a graph consisting of objects implementing the Executable interface.
- Executable. Implemented by all objects in a module graph. Product and Operation are the two most important subclasses.



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

4/22

- **Product.** A node in a module graph is always a product. Generally, a product corresponds to an element in CCXML or Voice XML. A product internally commonly contains 1 or more instances of Executable, ordered in a sequence.
- **Operation.** An operation is meant to perform a single, simple modification of the program state. There are several kinds of operations, all performing different actions when executed.

2.2 Compiler

This logical package contains the VoiceXML compiler and the CCXML compiler. The compilers translate various XML files into a structure more suitable for execution. VoiceXML and CCXML are the primary source languages, but other languages could also be integrated within the same structure.

The basic strategy for compilation is to use a `CompilerDispatcher` to find an appropriate `NodeCompiler` implementation for every XML element that is encountered during a walk of all XML nodes in the document. The `NodeCompiler` implementation found then emit a suitable executable representation of the XML node. In the terminology of the EE this executable is a tree-like graph of `Products` and `Operations`, where a `Product` acts as containing elements for `Operations`, and other `Products`. As an example of node-compilers, there is one `NodeCompiler` called `Assign`, able to compile the `assign` element into operations.

2.3 Runtime

The Runtime package contains classes that execute applications.

Execution of an application is performed similarly to how a stack based virtual machine works. The main difference to a general purpose CPU is that stack machines lacks registers and use the stack for all computations. A notable difference is that the runtime uses various methods and attributes to store program state, whereas a pure stack-machine would use raw memory.

Classes belonging to this package are:

- **Engine.** The Engine is a controlling module, traversing and executing objects implementing the Executable interface.
- **EngineStack.** During execution, the EngineStack contains a number of `StackFrame` instances, each stack frame describing the point of execution in a sequence of operations. A new stack frame signifies that an inner context of execution – in some sense – has been reached.
- **ValueStack.** Contains values produced and consumed by operations during the execution of an application.
- **Executable.** An interface implemented by all classes the engine can execute.
- **ExecutionContext.** The environment all operation executes in, contains a `ValueStack` and references to several other - language dependent – methods and classes



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

5/22

2.3.1 Event handling

An important part of the runtime is in handling events from internal and external sources. External events is received through an EventDispatcher interface published through the ApplicationExecution interface. Events received in this way is transported through an event bus implemented through chained EventStream objects. Eventually it is either delivered to a EventProcessor for further processing, or ignored.

2.4 Component interface

As all MAS components, the EE has internal and external interfaces. This logical package contains the interface other components would use for collaborating with the EE. Core interfaces is ApplicationManagement, ApplicationExecution and Application.

2.5 Platform Access

The Platform Access module contains methods to be used by the ECMA scripts to access the MAS platform methods.

These methods have functionality to access:

- Subscribers user profile
- Subscribers mailbox
- Send messages.
- Different medias for example voice and video prompts
- Number analysis
- Event reporting
- Configuration
- Utility functions

3 Function behavior

3.1 Starting an ApplicationExecution

This will basically setup the structure in section **Error! Reference source not found.** on page 3. Also, a LifecycleEvent.START will be internally sent.

1. start() is invoked on the new ApplicationExecution.
2. Create a CCXMLRuntime and it's associated objects, this runtime is known as the master runtime.
3. Create EventStream and attach it to the published EventDispatcher



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

6/22

4. Create a new RuntimeControl and connect it to the EventStream.
5. Start the CCXMLRuntime. This will locate the starting Product of the application to execute in the Engine, and fire a LifecycleEvent.START. As the engine is event driven, this will trigger it to start executing.

3.2 Compiling an application

Compiling an Application is performed by invoking the CCXML compiler on the CCXML documents, and the VoiceXML compiler on the VoiceXML documents in the application.

Both compilers will by a divide-and-conquer algorithm, divide the problem into smaller problems by compiling source files into modules, and modules into products and operations.

For examples of what the graph of executables may look like after compilation, see section 4 on page 18.

3.2.1 Relationships between compilation and execution

☞ The compilation is a process generating a program in form suitable for efficient execution.

All applications, and their associated Products and Operations must not contain mutable fields, as they are might be shared between different Engine instances.

3.2.2 Compiling a source file into a Module

The compilation is a recursive process that is performed element-by-element, with the help of NodeCompilers. As soon as a node compiler encounters another element in the element currently compiled, a new node compiler appropriate to compile that element is retrieved and control is passed to that compiler.

Consider the following VoiceXML-like notation:

```
<vxml>
  <audio src="welcome.wav" />
</vxml>
```

If there existed a Compiler for this notation, compiling the fragment would first invoke the node compiler able to compile the <vxml> element. During compilation, the node compiler would determine that there is an <audio> element embedded in the <vxml> element, and ask for the appropriate node compiler. In this case it is the audio node compiler, and then that compiler will be invoked.

Compilation of an element results in a product in the executable graph. The product has a sequence of executable instances, which are executed in order during execution.

A product may contain a mixture of operations and products; it's entirely dependent of the node compiler implementations. But it is considered good style that each XML element generates at most one product.



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

7/22

The entire compilation can consist of several passes, for VoiceXML two passes is made. The first normalizes noinput, help, nomatch, and error event handlers into regular catch tags; the second pass performs the actual code-generation. CCXML currently only makes one pass.

3.2.3 Operations

☞ **Operations are intended to be executed in the defined sequence**, this sequence is defined when compiling the product containing the Operations.

☞ **Operations are in many cases only meaningful together**. As an example, the first two operations in the sequence may push one value each onto the value stack, representing intermediate values, while a third operation may take two values from the value stack and perform an operation on the execution context based on these values.

☞ **An Operation does not know what operations that comes before or after it**. Certain operations assume that specific actions can be performed successfully on the ValueStack or with the ExecutionContext. The compiler must ensure that any such requirements are met.

3.2.3.1 Destructor part of a Product

The destructor is a separate sequence of executables in each product. These executables are used to ensure resource release or similar actions when the engine needs to unwind the stack.

3.2.3.2 Operation naming

Operations are named using a suffix notation describing their effect on the ValueStack. The general form is of the suffix is $_ [T [n | M] | Pn]$, where n denotes the number of values, T is take/pop, P is put/push, and M is all values until the sentinel value – here called “mark” – is found.

Consider the operation “plus_T2P”. This operation will take two values from the value stack, add them together - using regular addition - and push/put the result back onto the value stack.

An operation may use a fixed number of values from the value stack, or it may use all values up to the latest “mark”. The purpose of the mark is to be able to implement variadic functions, functions that can take any number of arguments.

Table 1, Sample operation suffixes

TM	Pop values from stack until a Mark is found
T3P2	Take three values, push two values

3.3 Execution

Execution is performed by the runtime module.



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

8/22

3.3.1 Relationships between Application and Runtime

Before execution of the application is started, one Runtime instance is created, this is the CCXML runtime, containing an Engine, and an ExecutionContext. If any other Runtime is to be started, this will be done through a DialogStart event. This event is handled by the class RuntimeControl which manages the lifecycle of all Runtimes except the master runtime.

3.3.2 Starting the execution

Start of execution happens when the someone outside the Execution Engine invokes start() on the IApplicationExecution object that was handed out when the service was loaded.

This will start the Engine appointed as the master engine. In the current implementation this is always a CCXML engine.

3.3.3 Engine algorithms

The Engine traverses the graph of Executables in the currently executed Module in **document-order, which is equivalent to a depth-first execution**. However, the engine has no knowledge of a Module as a concept or class. It simply executes the Products and Operations contained in the Module.

A reference to the current position in the executed graph is maintained as a "program counter". The following algorithm can serve as a coarse description of how the engine performs its work.

The Engine performs the execution using the following algorithm:

```
While not event-driven
  Pre-process
  Locate the current Executable
  Tell it to execute
  Post-process
  Check if there are incoming events
```

If the Engine has become event-driven it will pause, and only resume execution when an event is received. There are several cases that can lead to the engine becoming event-driven, but the most common is that the application has decided that it must wait for some external stimulus before it continues.

At some point, the Engine invokes the current Executable. This may imply one of two things:

1. If the current Executable is a Product, it is called, and a new StackFrame is pushed onto the EngineStack. The "program counter" is repositioned to the first Executable of the called Product.
2. If the current Executable is an Operation, it is invoked. What happens is depending on the executed Operation.

Two stacks are used by operations executing in the engine, The EngineStack and the ValueStack, but only the EngineStack is directly known to the Engine. The ValueStack is used by Operations, but not known to the Engine. The top-most stack frame describes the currently executing position in the graph. There are no application defined values in a StackFrame.



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

9/22

3.3.4 Program state

The program state is defined by the following entities:

- Current Module to execute.
- Program counter, i.e. the current Executable in the current Module.
- The EngineStack. This is a stack of StackFrame instances, where each StackFrame is a sequence of the current Operations to execute. That is, calling a Product implies copying all the Executables onto the new StackFrame. The reason for copying the “code” is in order to deal with event handlers of VoiceXML where it is stated that the event handler shall be treated like a local copy. However, because Executables are immutable, we only need to copy a reference to the list containing the Executables in question.
- The ValueStack. This is a stack of Value instances. A single Operation will push and pop Value instances to/from the stack.
- The Scopes published by the ScopeRegistry. This contains values and scopes created for and by the execution of ECMA scripts, and are handled by Rhino.
- Any state within the ExecutionContext

3.3.5 Invoking of a Product

From time to time, the Engine will encounter a Product in the Executable graph. The Engine will invoke execute on the Product, which in turn invokes call on the ExecutionContext, and this will cause an invocation of call on Engine.

The Engine will push a StackFrame onto the EngineStack at this point.

The Engine will remember which Product that caused this call, which is needed when and if the stack needs to be unwound.

Whenever a new frame is entered, the Engine will call the ExecutionContext which then stores the size of the ValueStack before this call. The size is stored in the Data member of the new StackFrame. Because of this we can also unwind the ValueStack when we unwind the EngineStack.

3.3.6 Executing an Operation

Whenever the Engine encounter an Operation in the Executable graph it will invoke the execute method on the Operation.

Operation is an interface, and there exists many implementations of this interface, currently more than 30.

It is common for operations to affect the ValueStack by storing or retrieving values. As described in 3.2.3 you can infer how and if an operation does alter the ValueStack by examining it's suffix.

3.3.7 Unwinding the stacks

At a goto, unwinding of the EngineStack and the ValueStack must take place.



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

10/22

For the EngineStack, this is done by going through each StackFrame in reverse order, invoking all destructor operations in that StackFrame until the sought unwinding point is reached.

Unwinding of the ValueStack is done in collaboration with the ExecutionContext by notifying the ExecutionEngine that a specific StackFrame has been reinstated.

3.3.8 Event handling

The Engine regularly checks if new Events have arrived. This is done between executions of each Operation. When executing destructors and a few other constructs, the check for new events is not performed. This exception is because handling events can alter the flow of control in the application running on the engine, and destructors, as an example must always be run to completion.

3.3.8.1 Event routing

Events within EE are transported by collaboration between several classes. The figure below illustrates a static view of an EE session with active CCXML and VoiceXML engines. Injectors and extractors filters events delivered through them. An Injector uses its filters to decide if the Event should be sent downstream and/or upstream towards VoiceXML or other dialogs. The upstream and downstream directions are defined in terms of upstream being from the CCXML master runtime towards a VoiceXML dialog, and downstream being the opposite direction. An extractor works in a similar manner, but uses its filters to decide if a particular event should be delivered to its consumer/peer.

In a sample scenario, an event is to be received from an external event-source, and ultimately delivered to an active VoiceXML dialog.

- External source delivers an event to the EventDispatcher interface of ApplicationExecution.
- EventDispatcher sends the event to all subscribed EventDispatchers. One of these subscribers is an Injector implementing the EventReceiver interface.
- The Injector examines its rules, and determines that the event should be sent both upstream, and downstream. Most injectors should probably always send events both up, and downstream. But there might be situations where this behavior is not desirable. In this case, we have no downstream to send to, but the Injector itself is oblivious to this fact.
- The CCXML EventStream examines its extractors, and determines that none of them is interested in the event. It then attempts to pass the event to the next upstream and downstream EventStream instance. In this case, only the upstream send will succeed since no downstream instances exist.
- The VoiceXML EventStream receives the event and examines its extractors. One, or several of the extractors rules matches the event, and it is then delivered to all matching extractors in order. If one of the extractors flags the event as delivered, no further delivery to other extractors will be done.
- An Extractor delivers the event to the EventProcessor

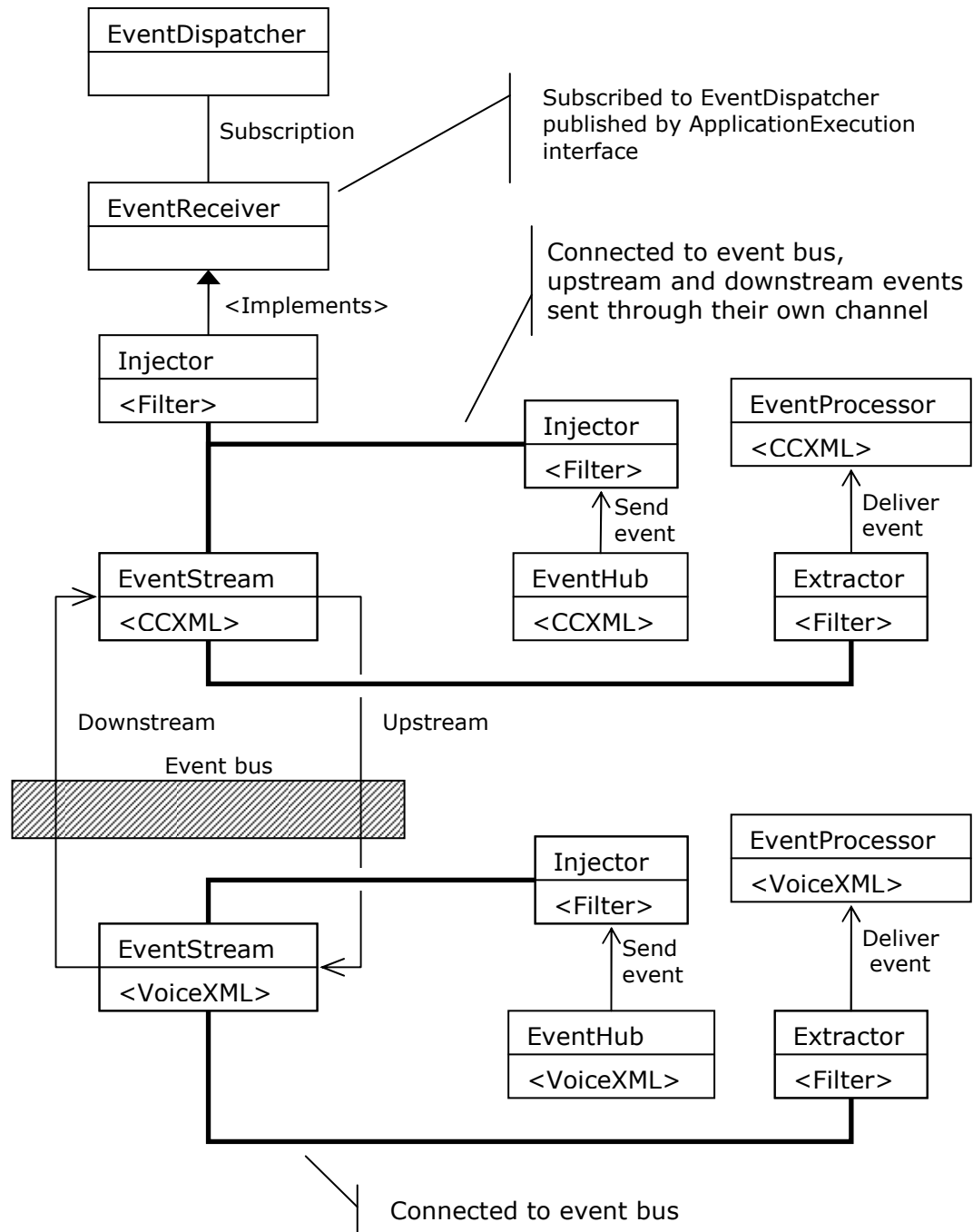


Figure 2, Event handling structure in EE with active CCXML and VXML documents

3.3.8.2 Event processing

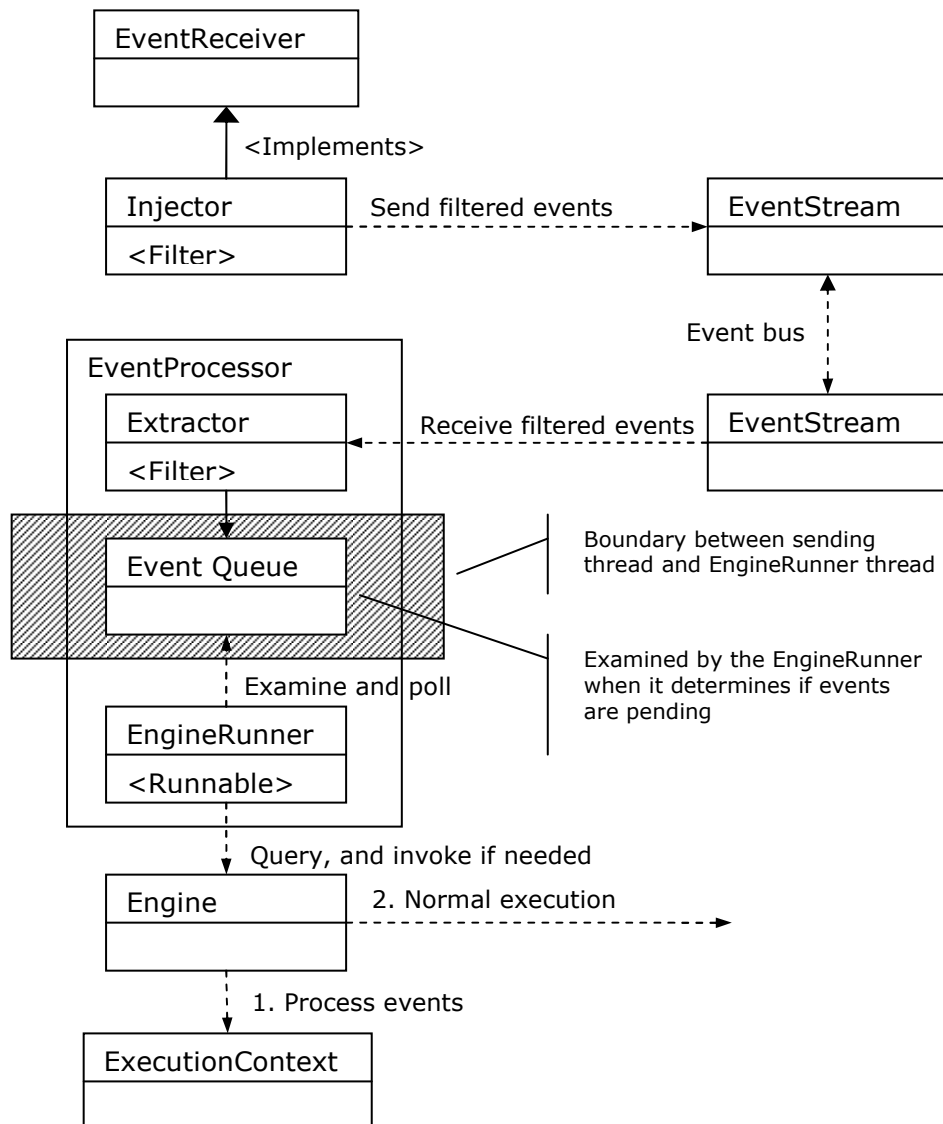


Figure 3, Event processing

3.3.9 Design principles of runtime classes

1. The Engine is largely ignorant of whether it executes a VoiceXML or CCXML application.
2. The Executable uses an ExecutionContext as an interface to the rest of the runtime classes.



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

13/22

3. The ExecutionContext should aggregate complex functionality not easily implemented by composing lists of simple operations and products. This might necessitate breaking up the ExecutionContext into several sub-interfaces or aggregations.
4. Invoking operations in interfaces outside EE shall be asynchronous, since this makes it easier to supervise the response. A synchronous invocation could block, and this would require more thread-code to do correctly. In the current incarnation of the platform access code, it's not asynchronous. This has already lead to a host of problems, some easier to fix than other.

3.4 Executing a VoiceXML subdialog

One of the more subtly complicated functions of the EE is in the execution of VoiceXML subdialogs. Each subdialog is to be executed as if it was an entirely new application, sharing nothing with its parent dialog except the connected telephony session and external event sources.

A rough outline of the steps needed when invoking a subdialog is as follows:

- Create a new runtime complete with Engine, ExecutionContext etc.
- Stop event delivery to all EventStreams which are ancestors to this runtime.
- Move any pending events from the parents EventProcessor to the subdialogs EventProcessor.
- Connect subdialog to the parents EventStream, disconnect the parent.
- Restart event delivery to EventStreams

After these steps are done, the subdialog is executing independently from the parent dialog. In fact, the parent is in a sense hibernating since it can only be woken up by a `com.mobeon.return` or `error.badfetch` event being sent directly to the parents EventProcessor, without passing through the EventStream. Normally this can only happen by the means of a `SendReturn_T` operation because the EventStream normally connected to the parents EventProcessor is still connected to the subdialog.

Returning from a subdialog is done as follows:

- Stop event delivery to all EventStreams which are ancestors to this runtime
- Shutdown the subdialogs runtime, this also reconnects the parents EventStream by invoking the shutdown hook prepared when the subdialog started.
- Move any pending events from the EventProcessor to the parents EventProcessor.
- Deliver `com.mobeon.return` to the parents EventProcessor
- Restart event delivery to EventStreams

Stopping external event delivery is done by locking the root EventStream so that no further events can be delivered, and no events will be lost. This is much easier



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

14/22

said then done. There are some changes that could be made to the EventStream implementation to make this easier. This could be achieved by adding an internal queue of matched, but not yet extracted events to each EventStream and by converting the extractor interface to a notify-and-poll interface. In this way only extraction need to be prevented, not notification. The drawback is that rules must be matched twice, but since the event delivery rate is quite low. The decreased code-complexity would more than compensate for that

3.5 Stability principles

- Detect failures as early as possible.
- Consider quick growth of number of threads as a problem. This is not currently implemented.

3.6 Run-time behavior

3.7 Platform Access

The ECMA scripts can use the platform access interface via two global objects named "mas" and "util". Ex:

```
mas.subscriberGetMailbox("161074");  
util.getCurrentTime("timezone");
```

3.7.1 Object identities

The Platform Access interface uses "identities" to communicate with the objects in the MAS platform. An identity is always assigned by the MAS platform and returned as the result of a method. All the identities are integers and the type of identity is defined by the name of the parameters, e.g. mailboxId and messageId.

Ex: The method *subscriberGetMailbox* returns an identity (mailboxId) for the user's mailbox. This identity can later be used in subsequent methods for example *mailboxGetByteUsage*.

3.7.2 Error Handling

Errors that occur during the execution of the methods are reported through the VoiceXML platform defined events. The following general errors are defined:

- error.com.mobeon.platform.profileread
- error.com.mobeon.platform.profilewrite
- error.com.mobeon.platform.mailbox
- error.com.mobeon.platform.numberanalysis
- error.com.mobeon.platform.datanotfound
- error.com.mobeon.platform.systemerror



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

15/22

When platform access detects an error an exception is thrown. The exception contains one of the above events and more information about the error. The execution of the script is terminated and the event can be received in the VXML document.

3.8 Platform Access

Platform access only catches a predefined set of exceptions thrown by the platform, and send those to the application as an event as documented in its corresponding JavaDoc. These Exceptions are "known" by the platform and a result of explicit error handling.

All other exceptions are handled by the Execution Engine in the same manner as it handles other exceptions (probably by terminating the session). Typically this is a NullPointerException.

3.9 Resource Locating

Some vxml tags have a src attribute that is an URI to an external resource that is located either on the disk or on an external server. The handling of the fetching of external resources is encapsulated in a special class ResourceLocator which is located in the *com.mobeon.masp.execution_engine.externaldocument* package.

The tags that currently support this feature are <audio>, <goto> and <subdialog>.

Two types of resources can be fetched, media and documents. Media is an audio or video file and document is a vxml file. The media is encapsulated in an IMediaObject. The document is compiled into a Module.

Both media and document is put into a cache after they have been fetched.

3.9.1 Cache

The resources are stored in a cache, key is the URI. The *shifto* cache library is used for the implementation of the actual cache code. The LFU cache algorithm is used which means that the least frequently used objects are removed if the cache is full.

Max size is set to 100 documents and 50 media objects. The max size is not configurable.

Each resource has a timestamp parameter which is set when the resource is added to the cache. VoiceXML defines a parameter "*maxage*" which is used to tell the server when to update (refresh) the data. EE checks this parameter against the timestamp on the resource and determines if it should be fetched again. Other VoiceXML attributes and properties available for caching management are not supported.

3.9.2 HTTP resource locating

If the URI starts with **http** the resource must be fetched via HTTP. This is encapsulated in a class `HttpResourceLocator` which is located in the `com.mobeon.masp.execution_engine.externaldocument.http` package.

It utilizes the java class `URLConnection` for the HTTP connections.

3.9.2.1 HTTP conditional get

HTTP1.1 specifies a special request header: "If-Modified-Since". If it is set the HTTP server checks that time against its internal time on the specific file that is requested. If not newer than the requested time it answers with "304 Not modified", and no data is sent in the response. EE uses this feature too and sends this header in the request, so even if the resource in the cache has aged it is not updated if the server responds with 304.

3.9.3 Sequence diagrams

Some specific scenarios are shown here:

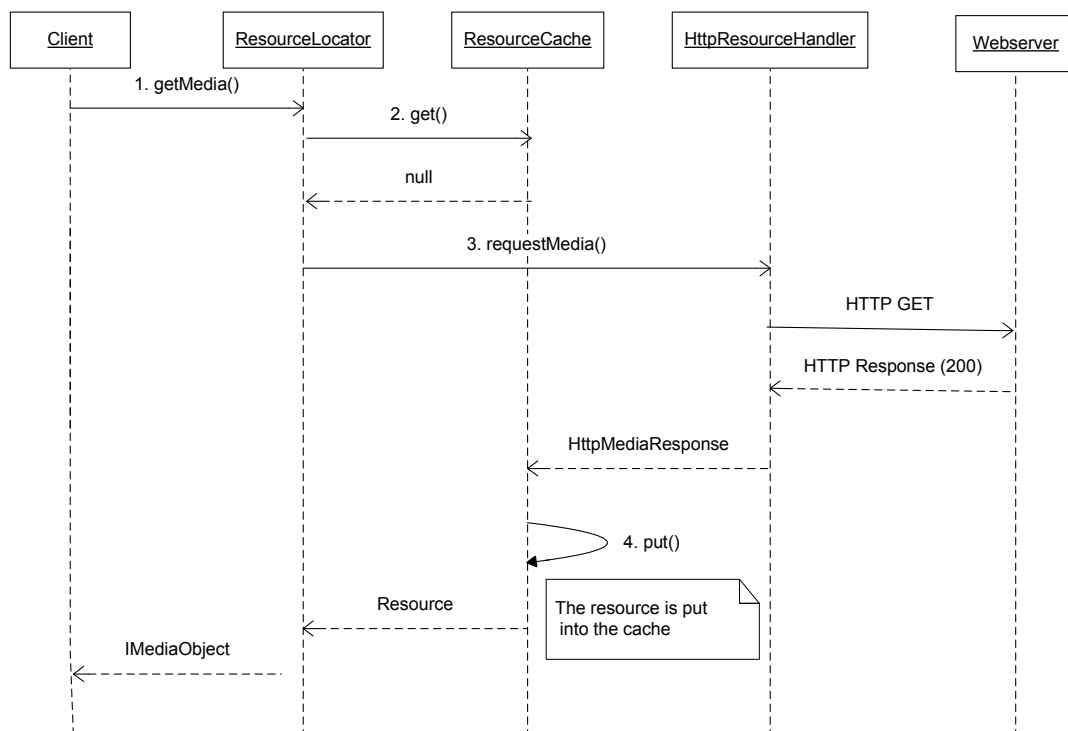


Figure 4, Data not found in cache and fetched from server.

1. The class that uses the ResourceLocator calls the `getMedia` method to fetch an `IMediaObject` to play.
2. The cache is checked, URI is used as key. The resource is not found.

3. Because the URI has scheme *http* the *HttpResourceHandler* class is used. It sends a HTTP request to the server specified in the URI. The server responds with 200 OK and the data is loaded from the response.
4. The resource is put into the cache; a timestamp is set on the object. At last the resource (*IMediaObject*) is returned to the caller.

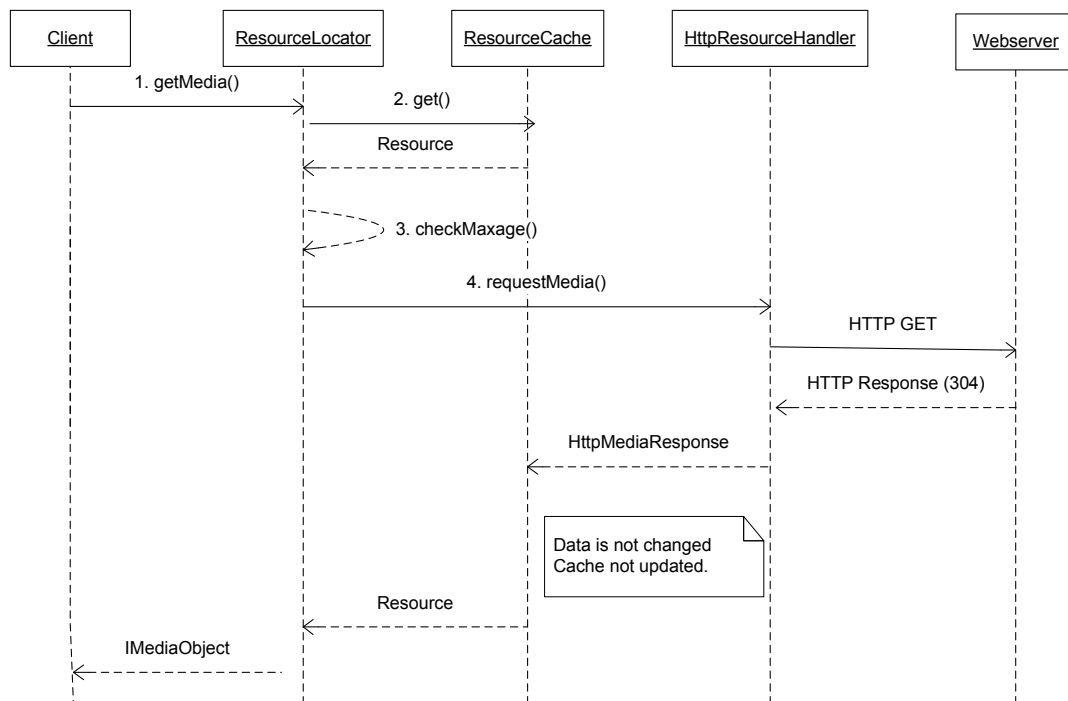


Figure 5, Data in cache has aged, but http server says it's valid.

1. The class that uses the *ResourceLocator* calls the *getMedia* method to fetch an *IMediaObject* to play.
2. The cache is checked, URI is used as key. A resource is found.
3. The timestamp on the resource is checked against the maxage parameter (if present). The resource has aged and must be refreshed.
4. Because the URI has scheme *http* the *HttpResourceHandler* class is used. It sends a HTTP request to the server specified in the URI. This time the "If-Modified-Since" header is set to the same time as the timestamp. The server responds with 304 Not Modified and no data are present in the response.
5. The cache is not updated in this case. At last the resource (*IMediaObject*) is returned to the caller.



4 Appendix A: Examples of graphs of Executables after compilation

Note: this section gives conceptual examples.

In reality, the set of Operations used may be different, and optimization passes during compilation may produce graphs that do not look like what you find in this chapter. The purpose of this section is to give conceptual understanding, and if you want to know exactly what the Executable graph looks like for a particular program, you are advised to compile the program and examine the debug output of the compilers.

The reason to only be conceptual here is to avoid maintainability problems as the source code of the compilers is changed.

However, you can be sure that execution of the Executable graphs you find here would produce valid results.

4.1 Description of the Operations used in this section

This section describes the example Operations used in this section. They may or may not exist in the real Execution Engine.

- newScope. This Operation introduces a new scope into the ECMA context.
- closeScope. This Operation closes a scope in the ECMA context, which basically means that all variables of that ECMA scope disappear.
- Text_P. This Operation pushes one piece of text onto the ValueStack.
- EvaluateECMA_TP. This Operation takes one Value from the ValueStack, evaluates it in the ECMA context, and pushes the result as a Value onto the ValueStack.
- LoadResource_T. This Operation takes one Value from the ValueStack, and loads it into the ECMA interpreter. For example, this Operation is able to load "foo.js".
- pushCall

4.2 Example 1

The following VoiceXML snippet:

```
<script expr = "'babba'+''.js'"/>
```

Could be compiled into the following Operations:

Text_P("'babba' + '.js'")

EvaluateECMA_TP().



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

19/22

4.3 Example 2

This sample is a CCXML document that is part of the automated testing, and test functionality for accepting an inbound call.

```
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0" xmlns="http://www.w3.org/2002/09/ccxml">

  <eventprocessor>

    <!-- Handle incoming call. -->
    <transition event="connection.alerting">
      <accept/>
    </transition>

    <!-- The call is fully connected. -->
    <transition event="connection.connected">
      <log expr="'Pass'"/>
      <exit/>
    </transition>

  </eventprocessor>
</ccxml>
```

To show how this actually is compiled, the compiler was set to generate .ops files, and imported the result. It was then adapted for presentation by shortening overly long names, and removing extraneous data only needed for debugging purposes. Some arguments were also removed, notably the call to RegisterHandler was simplified.

The left column is the generated program, and the right column is the annotated CCXML source.



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

20/22

Product (id='product_1891')

Constructor

NewScope ([ccxml])

Main

SendCCXMLEvent('ccxml.loaded', '')

Product (id='product_1893')

Main

Product (id='product_1894')

Constructor

RegisterHandler(

Event : connection.alerting

Target : product_1895)

RegisterHandler(

Event : connection.connected

Target : product_1896)

Main

ECMAVar('_state')

UseStateVariable('_state')

SetEventsEnabled(true)

Declarations

TruePredicate (id='product_1895')

Constructor

Atomic

SetEventsEnabled(false)

NewScope([transition])

Main

SetEventVar('_evt')

EventVar_P()

ConnectionAccept_T()

Destructor

Atomic

CloseScope()

SetEventsEnabled(true)

TruePredicate (id='product_1896')

Constructor

Atomic()

SetEventsEnabled(false)

NewScope([transition])

Main

SetEventVar('_evt')

Product (id='product_1897')

Main

Mark_P()

EvaluateECMA_P('Pass')

Log_TM()

Atomic

EngineShutdown(true)

SendCCXMLEvent('ccxml.exit',
'CCXML Exiting')

Destructor

Atomic

CloseScope

SetEventsEnabled(true)

<ccxml>

<eventprocessor>

Defines <transition
event="connection.alerting"/>

Defines <transition
event="connection.connected">

Declares <transition
event="connection.alerting">

<accept/>

.

</transition>

Declares <transition
event="connection.connected">

<log expr="'Pass'"/>

.

.

<exit/>

.

.

.

</transition>

</eventprocessor>

</ccxml>



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

21/22

5 Appendix B: 3PPs

5.1 Third-Party Products and Freeware

3PPName / Freeware Name	Version of the product/ freeware	Company	Used for	Delivered with the component	ECCN US/EU	Product No. and R-state
Spring	1.2.7	www.springframework.org . License: http://www.apache.org/licenses/LICENSE-2.0.html	Dynamic resolution of class dependencies.	Yes	EAR99	SWF0060 R1A
JavaBeans Activation Framework	1.0.2	Sun, http://java.sun.com/products/javabeans/glasgow/jaf.html . License: http://developers.sun.com/license/berkeley_license.html	Mimetype handling.		EAR99	SWF0007 R1A
Dom4j	1.6.1	www.dom4j.org . License: http://www.dom4j.org/license.html	Parsing VXML/CCXML files		EAR99	SWF0051 R1A
Rhino	1.6R2	http://www.mozilla.org/rhino/ , license: http://www.mozilla.org/MPL/	Execution of ECMA scripts		EAR99	SWF0055 R1A
Xpp	3-1.3.4.O	Indiana university, http://www.extreme.indiana.edu/viewcvs/~checkout~/XPP3/java/LICENSE.txt	Pull parser for parsing XML files.		EAR99	SWF0070 R1A



Approved: Magnus Björkman

Mobeon Internal

No: 3/FD-MAS0001 Uen

Copyright Mobeon AB
All rights reserved

Author: ERMKESE QDALO MPAZE QMIAN EJEFRID
Title: FD – Execution Engine

Version: B
Date: 2008-08-05

22/22

6 References

[1] <Document name>
<Document number>

7 Terminology

Term	Explanation
------	-------------