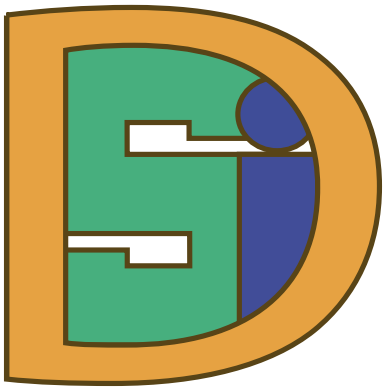

DSI IDL

Version 0.1

Harman/Becker Automotive Systems GmbH

Draft

DSI IDL: Version 0.1

Harman/Becker Automotive Systems GmbH

Becker-Göring-Straße 16, 76307 Karlsbad, Germany

Author:
Florentin Picioroaga Florentin.Picioroaga@harman.com

Person in charge:
Florentin Picioroaga Florentin.Picioroaga@harman.com
Publication date 21.12.2011

Draft

Revision History

Revision 0.1	December, 2011	FPicioroaga
Initial version		

Table of Contents

List of Figures

List of Tables

Document overview

DSI(Distributed Service Interface) is an inter-process communication (*IPC*) used in current HARMAN products.

This document is part of a documentation package comprising DSI and focuses only on the interface description language (*IDL*) used to formally describe application interfaces. Other parts of the documentation package are:

- inter-process communication (IPC) protocol describing the data serialization and the DSI protocol
- language binding, using DSI in different programming languages or frameworks e.g. HARMAN component based framework *MoCCA* (Modular Car Computing Architecture)
- Service Broker, creating an environment where applications can dynamically find services in the system

The document is organized in the following chapters:

1. Introduction

   In this chapter an overview of the DSI IPC is presented and the basic principles are explained

2. Interface Description Language

   The main chapter explaining the IDL features available in DSI.

1 Introduction

This chapter provides an introduction to the DSI IPC mechanism explaining the basic concepts that are implemented in the protocol and which an application needs to handle to connect to a DSI service.

1.1 Client/Server model

DSI IPC protocol is based on a server/client model where the server's interface, aka the service interface, is described formally in an *XML* based IDL and the client is the user of this interface. The DSI was designed to be used in a component framework (MoCCA), but is not restricted to that , where a component can implement and use several service interfaces and can be easily deployed in different application configurations.

1.2 Distributed communication

The DSI IPC was designed to be used in a distributed environment allowing communication for all possible deployment configurations of the client/server components :

- in-process communication where client/server components are deployed in the same process
- in-node communication where client/server components are deployed in different processes on the same CPU
- inter-node communication where client/server components are deployed in different processes on different CPUs

A consequence of this requirement is that currently, DSI supports three transport protocols: TCP/IP, Unix domain sockets and QNX message passing. An extension of using shared memory for inter-component communication deployed in the same process is provided currently by the internal HARMAN frameworks.

The following figure depicts three applications using DSI interfaces that are communicating .

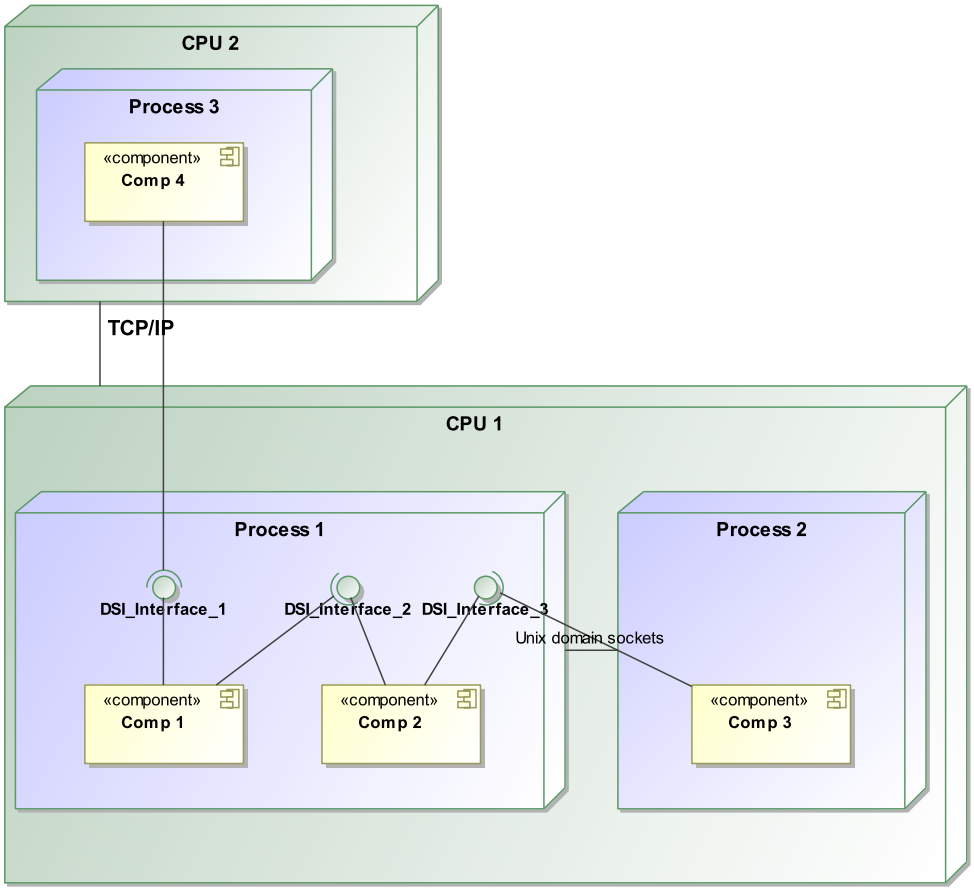

Figure 1.1. Distributed communication using DSI interfaces

1.3 Service discovery

Another important aspect when using DSI is the service discovery mechanism provided by the Service Broker. DSI applications register the implemented and required service interfaces with the Service Broker.
The Service Broker has the role to provide:

- a notification mechanism to inform applications about services that sign-in and sign-out.
- address information used by applications to establish communication over the DSI IPC.

The Service Broker can be used in a distributed system and can scale gracefully its infrastructure over many nodes. The naming service organization structure is similar with the DNS (Domain Naming Service) structure where each node in the network is serviced by a Service Broker and the Service Brokers are organized in a tree structure. A slave Service Broker will forward its unresolved requests and its information about local existing services to its parent, its master Service Broker.

The Service Broker is part of the DSI run-time environement and is available together with the DSI IPC package. The protocols used between DSI applications and the Service Broker and between the Service Brokers are described separately.

The following figure depicts how the applications use the service discovery infrastructure to establish a DSI communication.

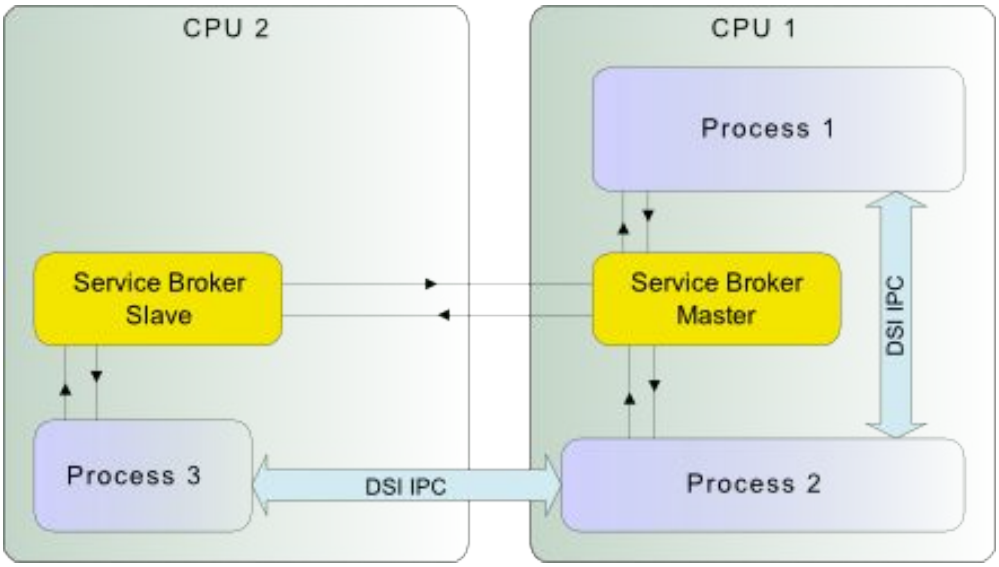

Figure 1.2. DSI run-time environment

1.4 Performance and footprint

One important requirement when designing DSI was to have good performance and small footprint. This is crucial for an automotive application where critical requirements related to start-up time or fast reaction times exist. Therefore, the DSI implementations are quite lightweight in terms of memory footprint (e.g. ~250KB for QNX) and CPU time (see IPC benchmark in GENIVI).

2 Interface Description Language (IDL)

2.1 Overview

This chapter describes the DSI IDL covering:

- the IDL specification
- an overview of the DSI IDL editor
- an overview of the code generation

2.2 Formal interfaces

One of the more powerful modern programming patterns that has arisen in the last years is the concept of formalized interfaces. These formalized interfaces have two essential properties:

- They support transparent communication across thread and process boundaries – in some cases even across hardware boundaries.
- They provide a formal separation of interface and implementation. In an application a formally defined interface may exist more than once and each instance could be an encapsulation of a different implementation.

The applications based on DSI use formal interfaces described in an XML based language.

2.2.1 Service interface inspired by the CORBA model

The service interface is an implementation of a very small *CORBA* subset, suitable for use in resource-poor embedded systems - a sort of "poor man's CORBA". Specifically, it provides the ability to define a formal definition for a programming interface. This characterization is valuable because it provides an immediate feel for the role that the service interface plays.

A dedicated code generator is used within the build process to create proxy and stub objects for all defined service interfaces.

A client that wishes to make use of a service interface may create a proxy object. A server that provides an implementation of a service interface must inherit from the stub class in an OO programming language (e.g. C++) or register callbacks to the stub in a procedural programming language, in C.

For further details regarding code generation please see the appropriate chapter.

2.2.2 The DSI IDL concepts

Before going to describe the XML syntax of the DSI IDL in detail the main concepts will be presented: data types, methods to exchange data types and error handling.

2.2.2.1 Data types

An *API* will need the possibility to describe information to be transferred between applications. The DSI IDL is similar with other static languages where the structure of the information is known at compile time and provides a set of primitive data types and the possibility to aggregate the primitive data types in more complex structures.

Because out of the interface IDL definition code in a standard programming language will be generated the data types are strongly oriented at the C++ programming language way of handling data types.

Primitive data types

For an IPC mechanism it is important to define the byte order or the alignment. However, on the logical level, in the IDL these facts are not important but only the size of the data type is important because it defines how much information can be transmitted. The byte order and alignment can be freely chosen by the IPC transporting the data.

Integer Data Types

The integer types literals can be used as in C++ accepting the same syntax, e.g. using the decimal, octal, hexadecimal base notation.

Type	Size [bytes]
byte/char	1
signed/unsigned short	2
signed/unsigned int	4
signed/unsigned long long	8

Table 2.1. Size for integer data types

Floating Point Types

As for the integral types, the floating types literals use the C++ syntax, e.g. `1.23`, `.23`, `1.`, `1.2e10`, `2.0f`, etc.

Type	Size [bytes]
Float	4
Double	8

Table 2.2. Size for floating point data types

Boolean Type

A boolean data type is available and can have as values: `true` and `false`.

Enumeration Type

An enumeration is a data type that can hold a set of values specified by the user. Once defined, an enumeration is used very much like an integer type.

String Type

A string type with dynamic size is available where the content is encoded in UTF-8.

Buffer Type

A contiguos region of physical memory storage. A Buffer implementation can be different than a Vector of bytes implementation, the contiguity is not guranteed for the later case.

Compound types

The compound types can be seen as containers of primitive data types.

Structure type

An aggregate of elements of arbitrary types can be used with the `Structure` data type. Using this data type more complex data types can be built on base of the primitive types.

Vector Type

An aggregate of elements having the same data type with a dynamic size can be used in DSI IDL.

Variant

A variant is similar with the `union` data type in C++. It is a placeholder to contain just one element from a defined set of different data types.

Map

A map is a collection of (key,value) pairs where a key appears only once in the collection.

2.2.2.2 Exchanging data

Having possibility to encode information in the available data types, the IDL provides also mechanisms to exchange this information.

Attributes - interface state information

Attributes can be used to store state information and are different to events or signals which encode stateless information that is lost as soon as the signal is emitted. The publish/subscribe messaging pattern is provided to the client to retrieve the state information. A client will register to the server to be notified on changes in the values of the data attributes. When the data changes on the server side the client will get the new value of the attribute.

Data attributes and the parameters of request and response methods may have any desired type. Both built-in and user-defined types may be used but several restrictions apply. One restriction is that pointers or references are not supported. If a pointer or reference must be transported, it is the responsibility of the developer to package it as a data element within some class. He then also carries the burden of deciding how this data element can be meaningfully transported across an address space boundary.

Informations or events - interface stateless information

Informations in DSI IDL can be used by the server implementing an interface to send stateless information or events. The clients can notify on available informations to get these events.

An information can have any number of parameters and also can be overloaded, the same information can exist with different signatures i.e. accepting different kind of parameters .

Request methods

The request methods allow a client to trigger actions on the server side. Depending on the underlying IPC mechanism the request method can be synchronous (block-

ing) or asynchronous (non-blocking). The DSI IPC protocol provides asynchronous calls.

A request method can accept any number of parameters. The server will process the request and can send back an answer in form of a response method.

Request methods can be overloaded, the same request can exist with different signatures i.e. accepting different kind of parameters. The designer of the interface can define for the request parameters default values which are optional to be provided by the client.

A request can be coupled to a response or can be of kind "fire and forget" with no response. When a request is coupled to a response the server must send back the coupled response method.

Response methods

The response methods can be used by the server as reply to a request method.

A difference to other IDLs is that in DSI other clients than the one that has triggered an action on the server over a request can notify on all the response methods and can get themselves these response. So the other clients can hear all the output traffic a server it does.

Another difference is that a server may send several responses as result of a client's request or may send the same response method to different request methods.

A response method, like a request method, can have any number of parameters and also can be overloaded.

Register methods - server side filters

Register methods can be seen as a mechanism to apply server side filters on the data the server sends to its clients over the information methods.

The server provides parametrized filters and the clients can customize these filters. When changes are sent by the server, the server will first filter the data and then send it only to the clients for which the filter applies. The clients can dynamically register/unregister to the server to make use of server filters.

This mechanism was introduced in order to improve performance in the system and avoid large traffic of data that will not be used on the client side.

Example:

Preconditions:

- the server can have for the power voltage of the system an attribute `voltage` but also an information `informationVoltage(`*`int voltage`*`)`

- the server provides for the power voltage information a register method, a filter with customizable parameters for the voltage conditions: `registerVoltage-Filter(int underVoltage, , int overVoltage)`
- a client monitoring critical voltage conditions is interested for under-voltage and over-voltage conditions.

Solutions:

1. the client can notify to get permanent updates of the power voltage attribute. In this case will get all updates including the normal range
2. the client can use the `registerVoltageFilter` with its values and will get from the server the `informationVoltage` with the current voltage only when the `voltage` is less than *`underVotage`* or greater than *`overVoltage`* specified values.

2.2.2.3 Error handling

Error handling does not have a dedicated support in the DSI IDL. Currently it is up to the interface designer to provide responses with error codes as parameters indicating if the request was processed successfully or not.

However, the DSI IPC implementation has a dedicated error handling.

For example the attributes can be invalidated from the server side, that means the clients will receive special events indicating that the attribute value can not be computed currently by the server.

Other example will be that a server can not process a request as it is busy processing the same request from other client. In this case the client will get a notification signalling the busy state of the server.

If the interface designer is defining an enumeration with the name `Error` then the code generator will generate an additional error handling that can be used on the server side to signal a predefined error code and on the client side to get it.

2.2.3 The IDL language

The DSI IDL is based on XML. Every interface is described in its own file and an XML schema is provided to check the validity of the file. The XML schema and an Ecore model are provided along with the IDL documentation package.

An example of a DSI IDL interface that covers the main concepts is provided too.

Note

It is important to know that the DSI IDL is a subset of another IDL used internally by Harman the HBSI (Harman Becker Service Interface). The DSI IDL does not use all

the features of the HBSI but for completion the elements of the HBSI will be mentioned here, too.

A common information available for many XML nodes and described only for the DSI node is composed from: `Name`, `Description`, `Deprecation`, `Hint` and `ID`. The `ID` is used just to have a faster code generator.

2.2.3.1 The `DSI` service interface

The XML node `DSI` contains information regarding the service interface like name, version, etc. and containers with additional elements available in the interface like data types, methods, attributes, etc. The interface information is described below the containers are described in the following sections.

`Name`

The name of the interface defined in the IDL file. The name will identify all implementations of the interface in the service discovery directory. Additionally the implementations will set an instance name, aka role name, to distinguish from other implementations of the same interface.

`Description`

Documentation of the interface. This can be used by the generator to produce Doxygen documentation directly in the generated source code.

`Deprecated`

Flag used to inform the interface clients that the interface is in a deprecation phase. The interface should not be used anymore.

`Hint`

In case the interface is marked as deprecated this field can be used to provide additional information about the deprecation reason or hints about other interface to be used in order to get the same functionality.

`Extern`

Flag indicating if the data type definitions are all DSI capable and can be therefore sent across platforms.

An application has the possibility to define its own data types in non DSI IDL files e.g. in C++ header files, or to use directly the API of a 3rd party software defined in these programming language specific files. These data types can be used only in-

ternally in the application, between different components deployed in the application.

`Abstract`

Flag to mark if the interface can be used directly or it has to be used over a specialization. The concept is similar with abstract classes in OO languages.

`Version`

The service interfaces have all version numbers in a form <major>.<minor>.

The general rules when defining the version number are:

- A change in the major version number (in the first position) signifies a change that will break compatibility. This means that a method or attribute has been deleted or that the signature of a method or attribute has been changed.
- A change in the minor version number (in the second position) signifies a change that doesn't break compatibility. An example would be when a method or attribute is added.

Given these rules, deciding if a client of one interface version can communicate with the server implementing another version, two conditions have to be satisfied:

- The major versions must be identical: MajorVersion(client) == MajorVersion(server)
- The minor version of the server must be greater than or equal to the minor version of the client: MinorVersion(server) >= MinorVersion(client)

`BaseInterface`

An interface can inherit from other interface - the base interface. The current DSI IPC implementation allows only extending of enumerations with new ids. No polymorphism is supported for the moment.

`Includes`

Additional IDL files can be included in the interface. These can contain data type definitions.

2.2.3.2 `DataTypes`

Container with data types defined or included from external files in the interface. The container is a sequence of `DataType` entries.

DataType

The primitive and compund data types supported in DSI IDL are: `Boolean`, `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Float`, `Double`, `String`, `Buffer`, `Variant`, `Vector`, `Map`.

A user defined data type has a name and can have, as the interface, information about: `Description`, `Deprecation`, and `Hint`.

Additionally the following information can be also provided: `Kind`, `Namespace`, `Include`, `Container`, `Fields` and `BaseType`.

Kind

The kind of a data type can be: `Structure`, `Streamable`, `Typedef`.

A `Structure` data type can be used to define a compound data type.

A `Typedef` data type is an alias for other data type.

A `Streamable` data type is one imported from an external non DSI IDL file which has already defined streaming operators (not relevant for DSI IDL).

Namespace

A data type that is imported in the current interface can be available under a `Namespace`.

Include

The data type is imported from an external file provided by this filed.

Container

The DSI IDL provides some standard containers which are: `Vector`, `Variant` and `Map`. These were already described in a previous section.

Fields

In case a data type is a compound type the elements of the structure are included in a `Fields` container.

Each `Field` XML node can provide information about: `Name`, `Description`, `Deprecation`, `Hint` and `Type`. The type of the filed has to be defined in the interface or imported from an external file.

Additionally, a field can have a default value defined over the `DefaultValue` XML node.

`BaseType`

If the data type is of kind `Typedef`, an alias, then the `BaseType` represents the original data type name.

2.2.3.3 Enums

This is a container where all the enumerations used by the interface are defined.

`Enum`

A enumeration has information such: `Name, Description`, `Deprecation` and `Hint`. The values for the enumeration can be defined over the XML node `EnumID`.

The `EnumID` can also contain information about: `Name`, `Description`, `Deprecation` and `Hint`. Additionally an integer value can be also specified for the enumeration id.

2.2.3.4 `Methods`

The `Methods` container defines all available methods of the interface.

`Method`

The `Method` XML node contains general information about: `Name, Description`, `Deprecation` and `Hint`.

A `Method` can be of type: `Request`, `Response`, `Information`, `Register` and `UnRegister`. All these are explained in the DIS IDL concepts section.

The `Request` methods can be coupled with a `Response` method over the XML node `Response`.

The parameters of the method are defined in the `Parameters` container.

Every parameter provides information like: `Name, Description`, `Deprecation` and `Hint`. A method parameter can have also a default value specified over the `DefaultValue` XML node. Such a method parameter is optional to be provided by the clients of the interface.

2.2.3.5 Attributes

The Attributes node is a container for all state information provided by the interface.

Attribute

An attribute is defined by information like: Name, Description, Deprecation and Hint.

The Type of the attribute refers to a data type available in the DataTypes container.

The Notify information can be:

- Always: the clients will receive the value of the attribute every time the server updates it
- OnChange: the clients will receive the values of the attribute only when the server changes it with a new value
- Partial: this applies only for the Vector data type. In this case the server can update only a contiguous part of the vector. This part can be replaced, deleted or new elements added in the vector.

2.2.3.6 Constants

The Constants node is a container for all kind of constants defined in the interface.

Constant

A Constant node has among general information like: Name, Description, Deprecation and Hint also a Type information and the constant stored in the Value XML node.

2.3 The service interface editor

Modifying the XML based definition of a DSI interface is error-prone when working directly with the XML content, therefore a dedicated editor is used to assist the user in this task. The service interface editor is an Eclipse plugin and at HARMAN is integrated together with other Eclipse plugins in an IDE, the MoCCA IDE. The plugin assist the user not only on modifying the service interface but also in doing additional validation tasks, e.g. detecting changes that must be reflected in a service interface version change.

The following figures show some snapshots when working with the service interface editor.

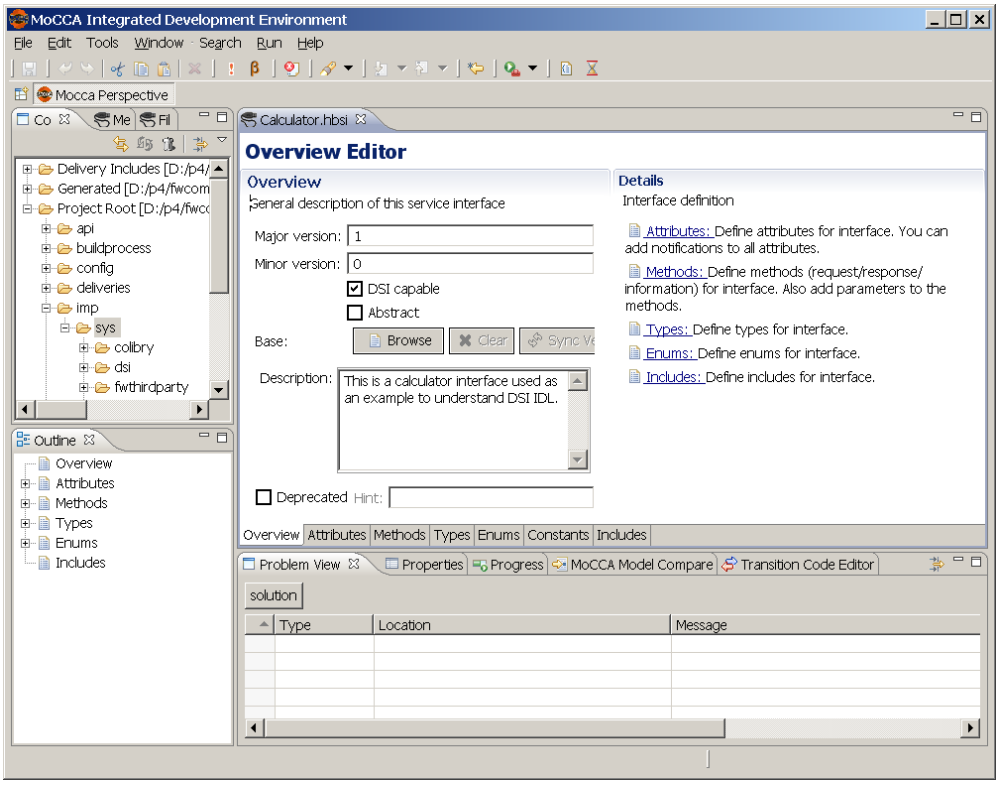

Figure 2.1. Service interface, overview.

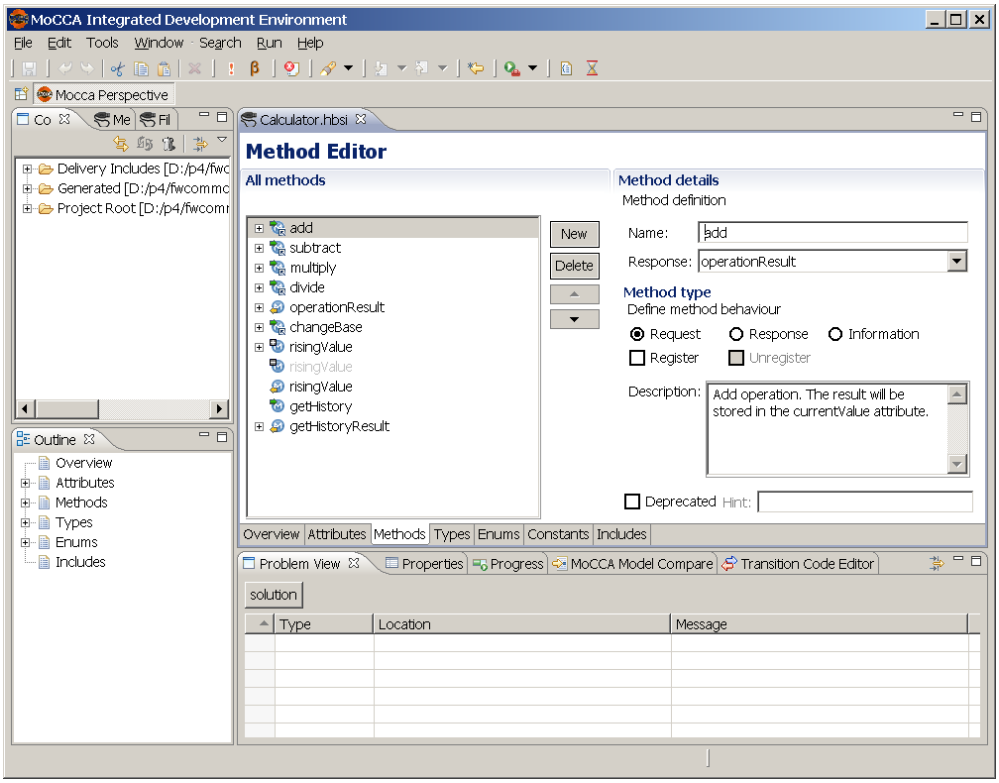

Figure 2.2. Service interface, method editor.

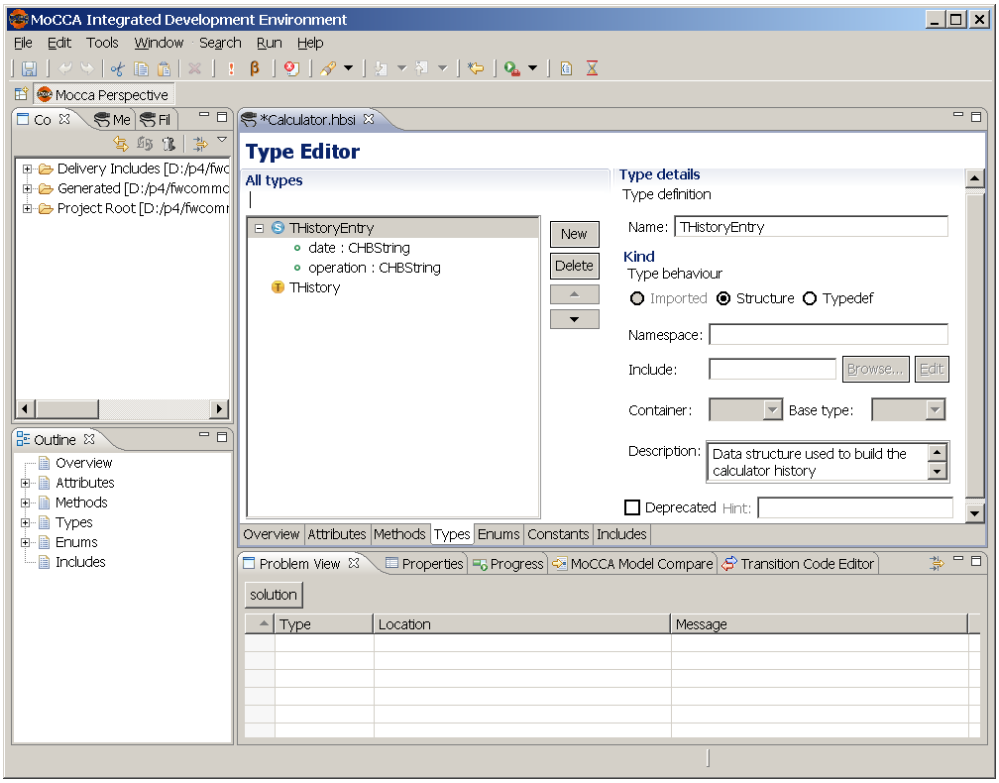

Figure 2.3. Service interface, types editor.

2.4 Code generation

The DSI interfaces may be imported and freely used by client applications. Support libraries and code generators are provided that allow DSI service interfaces to be imported and exported from within:

- a Java run-time environment
- a C++ environment
- a C environment

The DSI IPC glue code that brings together the server/client software is composed from two parts:

- a DSI library providing an implementation for the DSI IPC concepts
- a code generation part that generates out of the DSI IDL code that uses the DSI library and provides abstract concepts, easy to be used by the server/client software.

In the following section there will be presented an overview about the code generation part.

2.4.1 Service interface as a 'wrapper'

Every service interface that is defined can be viewed as a 'wrapper' around an implementation. The interface provides the formal definition of the mechanisms to be used when accessing the implementation. The interface mechanism provides a 'stub' class that must be used as the base class from which the implementation inherits. It also provides a 'proxy' class that can be instantiated in any application. To use the 'wrapped' implementation, a client must obtain a proxy. The client only sees the proxy, which will in turn ensure that all requests are correctly forwarded to the implementation.

This idea has several consequences. One is that the location of the implementation is invisible to the client. The implementation can be in the same address space with the client, in other process on the same platform (operating system and CPU) or on completely other platform.

In the same way, the service interface provides no information to the implementation about the origin of a request (which client and/or how many clients in total). Requests from a client are completely anonymous from the point of view of the application software.[1]

The following figure shows the interaction between the code generated out of the DSI IDL service interface and the client/server implementation.

[1]Of course the source of a request is known within the service interface glue code. This is required so that the service interface can send the response back to the originator of the request.

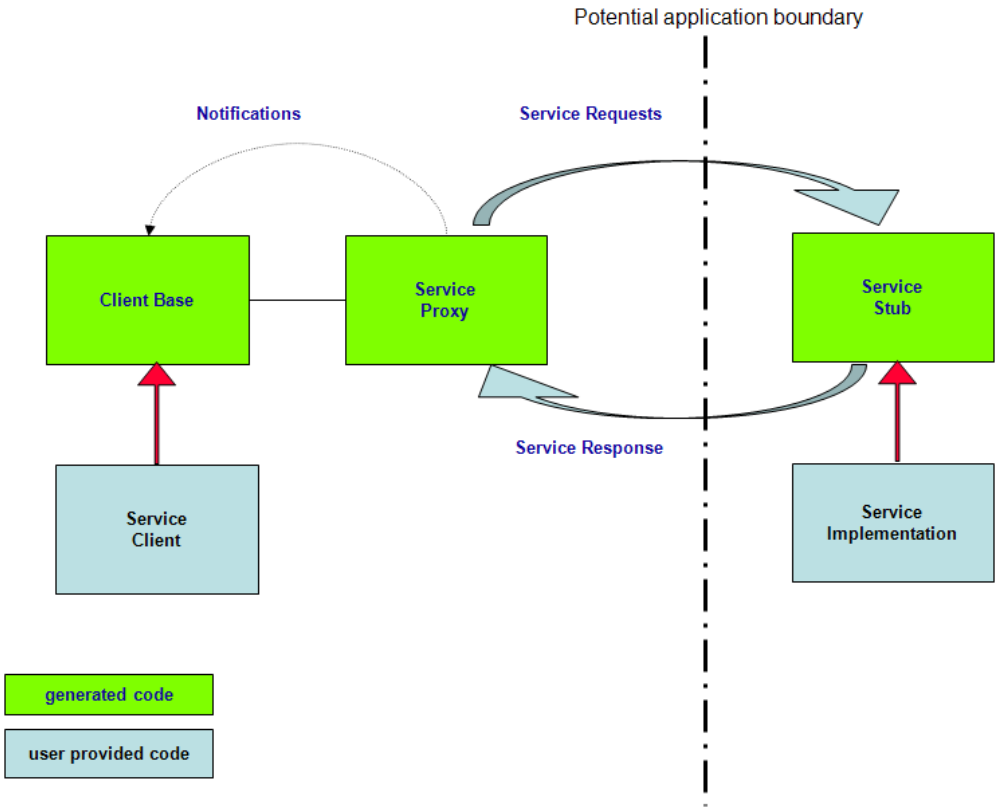

Figure 2.4. Code generation overview for DSI IDL

The code generation is currently based on the Eclipse JET solution for Model-Driven Development (MDD). Further implementations might move to the new Eclipse MDD solution Xpand / Xtext.

Appendix A. DSI IDL example

Revision History

Revision 0.1	14.December, 2011	FPicioroaga
Initial version		

A.1 Description

This is a simple example where a simple calculator interface is defined showing the main DSI IDL elements:

- data types: enumerations, a simple structure using a container
- attributes
- different method types: request, response, information and register

The example is also available as a separate file.

A.2 Calculator.hbsi

```
      <DSI>
  <Name>Calculator</Name>
  <ID>1</ID>
  <Description>This is a calculator interface used as an example to understand
DSI IDL.</Description>
  <Extern>true</Extern>
  <Version>
    <Major>2</Major>
    <Minor>0</Minor>
  </Version>
  <DataTypes>
    <DataType>
      <Name>THistoryEntry</Name>
      <ID>31</ID>
      <Description>Data structure used to build the calculator history
        of operations.</Description>
      <Kind>Structure</Kind>
      <Fields>
        <Field>
          <Name>date</Name>
          <ID>32</ID>
          <Description>The date the operation was executed on the
            calculator.</Description>
          <DefaultValue />
          <Type>String</Type>
        </Field>
        <Field>
          <Name>operation</Name>
          <ID>33</ID>
          <Description>The operation as string.</Description>
          <DefaultValue />
          <Type>String</Type>
        </Field>
      </Fields>
    </DataType>
```

```
  <DataType>
    <Name>THistory</Name>
    <ID>34</ID>
    <Description>The history of operations executed on the calculator.
    </Description>
    <Kind>Typedef</Kind>
    <Container>Vector</Container>
    <BaseType>THistoryEntry</BaseType>
  </DataType>
</DataTypes>
<Enums>
  <Enum>
    <Name>EBase</Name>
    <ID>35</ID>
    <Description>Numeric base the calculator can work with.
    </Description>
    <EnumIDs>
      <EnumID>
        <Name>BASE_UNDEFINED</Name>
        <ID>39</ID>
        <Description>Value used to detect an uninitialized parameter
          sent by the client.</Description>
        <Value>0</Value>
      </EnumID>
      <EnumID>
        <Name>BASE_DECIMAL</Name>
        <ID>36</ID>
        <Description>The decimal base.</Description>
        <Value>1</Value>
      </EnumID>
      <EnumID>
        <Name>BASE_HEXADECIMAL</Name>
        <ID>37</ID>
        <Description>The hexadecimal base.</Description>
        <Value>2</Value>
      </EnumID>
      <EnumID>
        <Name>BASE_BINARY</Name>
        <ID>38</ID>
        <Description>The binary base.</Description>
        <Value>3</Value>
      </EnumID>
    </EnumIDs>
  </Enum>
  <Enum>
    <Name>EResult</Name>
    <ID>40</ID>
    <Description>Possible results for calculator operations.
    </Description>
    <EnumIDs>
      <EnumID>
        <Name>OP_SUCCESS</Name>
        <ID>41</ID>
        <Description>Last request on calculator was successfully
          processed.</Description>
        <Value>0</Value>
      </EnumID>
      <EnumID>
        <Name>OP_ERROR</Name>
        <ID>42</ID>
        <Description>Last request on calculator was not properly
          processed. No further information provided about the cause
          of the failure.</Description>
        <Value>1</Value>
      </EnumID>
      <EnumID>
        <Name>OP_OUT_OF_BOUNDS</Name>
```

```
        <ID>43</ID>
        <Description>The result of the last computation is too large.
        </Description>
        <Value>2</Value>
      </EnumID>
      <EnumID>
        <Name>OP_NULL_DIVISION</Name>
        <ID>44</ID>
        <Description>A division with null leads to this error.
        </Description>
        <Value>3</Value>
      </EnumID>
    </EnumIDs>
  </Enum>
</Enums>
<Methods>
  <Method>
    <Name>add</Name>
    <ID>8</ID>
    <Description>Add operation. The result will be stored in the
      currentValue attribute.</Description>
    <Type>Request</Type>
    <Response>operationResult</Response>
    <Parameters>
      <Parameter>
        <Name>op1</Name>
        <ID>9</ID>
        <Description>The first operand to be added.</Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
      <Parameter>
        <Name>op2</Name>
        <ID>10</ID>
        <Description>The second operand to be added.</Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
    </Parameters>
  </Method>
  <Method>
    <Name>subtract</Name>
    <ID>11</ID>
    <Description>Subtract operation.</Description>
    <Type>Request</Type>
    <Response>operationResult</Response>
    <Parameters>
      <Parameter>
        <Name>op1</Name>
        <ID>12</ID>
        <Description>The operand to be subtracted from.</Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
      <Parameter>
        <Name>op2</Name>
        <ID>13</ID>
        <Description>The value to be subtracted.</Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
    </Parameters>
  </Method>
  <Method>
    <Name>multiply</Name>
    <ID>22</ID>
    <Description>Multiply operation.</Description>
```

```
    <Type>Request</Type>
    <Response>operationResult</Response>
    <Parameters>
      <Parameter>
        <Name>op1</Name>
        <ID>23</ID>
        <Description>The first operand to be multiplied.</Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
      <Parameter>
        <Name>op2</Name>
        <ID>24</ID>
        <Description>The second operand to be multiplied.
        </Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
    </Parameters>
  </Method>
  <Method>
    <Name>divide</Name>
    <ID>25</ID>
    <Description>Divide operation having as first operand the
      currentValue and the second operand as parameter.</Description>
    <Type>Request</Type>
    <Response>operationResult</Response>
    <Parameters>
      <Parameter>
        <Name>op1</Name>
        <ID>26</ID>
        <Description>The first operand to be divided from.
        </Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
      <Parameter>
        <Name>op2</Name>
        <ID>27</ID>
        <Description>The divider operand.</Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
    </Parameters>
  </Method>
  <Method>
    <Name>operationResult</Name>
    <ID>19</ID>
    <Description>Common information indicating the result of the last
      request that does a computation on the calculator: add,
      substract.</Description>
    <Type>Response</Type>
    <Parameters>
      <Parameter>
        <Name>result</Name>
        <ID>20</ID>
        <Description>The result of the operation.</Description>
        <Type>Double</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
      <Parameter>
        <Name>error</Name>
        <ID>21</ID>
        <Description>Error code for the operation.</Description>
        <Type>EResult</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
```

```
    </Parameters>
  </Method>
  <Method>
    <Name>changeBase</Name>
    <ID>14</ID>
    <Description>Change the GUI representation numerical base for
      input/output. On success the base attribute will be changed.
    </Description>
    <Type>Request</Type>
    <Parameters>
      <Parameter>
        <Name>newBase</Name>
        <ID>15</ID>
        <Description>The new base.</Description>
        <Type>EBase</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
    </Parameters>
  </Method>
  <Method>
    <Name>risingValue</Name>
    <ID>4</ID>
    <Description>Using filters on the server side. The client can
      register to get an information when the currentValue gets over
      the highWatermark parameter.&#13;
      The filter is applied on the
      server, this avoids traffic i.e. sending
      all the values and
      filter them on the client side.</Description>
    <Type>Register</Type>
    <Parameters>
      <Parameter>
        <Name>highWatermark</Name>
        <ID>5</ID>
        <Description>In case the currenValue rise above this watermark
          the clients will receive the risingValue information.
        </Description>
        <Type>Int32</Type>
        <IsDefault>false</IsDefault>
      </Parameter>
    </Parameters>
  </Method>
  <Method>
    <Name>risingValue</Name>
    <ID>6</ID>
    <Type>UnRegister</Type>
  </Method>
  <Method>
    <Name>risingValue</Name>
    <ID>7</ID>
    <Description>This information is sent while the high watermark is
      exceeded.</Description>
    <Type>Information</Type>
    <Parameters>
       <Parameter>
          <Name>value</Name>
          <ID>48</ID>
          <Description>The current value.</Description>
          <Type>Int32</Type>
          <IsDefault>false</IsDefault>
       </Parameter>
    </Parameters>
  </Method>
  <Method>
    <Name>getHistory</Name>
    <ID>16</ID>
    <Description>Get the history of operations run on the calculator.
```

```
        An example that uses a compound data type.</Description>
      <Type>Request</Type>
      <Parameters>
         <Parameter>
            <Name>numberOfEntries</Name>
            <ID>45</ID>
            <Description>The last entries from the history to get. If null then
the whole history is retrieved.</Description>
            <Type>UInt32</Type>
            <IsDefault>true</IsDefault>
            <DefaultValue>0</DefaultValue>
         </Parameter>
      </Parameters>
    </Method>
    <Method>
      <Name>getHistoryResult</Name>
      <ID>17</ID>
      <Description>The result of the getHistory request.</Description>
      <Type>Response</Type>
      <Parameters>
        <Parameter>
          <Name>history</Name>
          <ID>18</ID>
          <Description>The history of the operations.</Description>
          <Type>THistory</Type>
          <IsDefault>false</IsDefault>
        </Parameter>
      </Parameters>
    </Method>
    <Method>
       <Name>historyChanged</Name>
       <ID>49</ID>
       <Description>The history changed because a new request was processed by
the server.</Description>
       <Type>Information</Type>
       <Parameters>
          <Parameter>
             <Name>size</Name>
             <ID>50</ID>
             <Description>The current size of the history.</Description>
             <Type>UInt32</Type>
             <IsDefault>false</IsDefault>
          </Parameter>
       </Parameters>
    </Method>
  </Methods>
  <Attributes>
    <Attribute>
      <Name>currentValue</Name>
      <ID>2</ID>
      <Description>State information, the result of the previous
        operation executed on the calculator.</Description>
      <Type>Double</Type>
      <Notify>OnChange</Notify>
    </Attribute>
    <Attribute>
      <Name>base</Name>
      <ID>3</ID>
      <Description>The base to be used on the GUI interface: user input,
        calculator output.</Description>
      <Type>EBase</Type>
      <Notify>OnChange</Notify>
    </Attribute>
  </Attributes>
</DSI>
```

Glossary

A

API — Source code based specification intended to be used as an interface by software components to communicate with each other. May include specifications for routines, data structures, object classes, and variables.

C

C++ — Statically typed, free-form, multi-paradigm, compiled, general-purpose programming language.

CORBA — Standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.
See Also Common Object Request Broker Architecture.

D

DSI (SI) — The distributed service interface DSI in version 2 or greater is an inter-process communication protocol between several applications running on multiple platforms. DSI implementations exist for different operating systems like QNX, Linux or Windows, having different programming language bindings like C++, C and Java. As transport protocols DSI can use TCP, Unix domain sockets and QNX message passing.
See Also distributed service interface, DSI2, SI.

I

IDL — Specification language used to describe a software component's interface. IDLs describe an interface in a language-neutral way, enabling communication between software components that do not share a language for example, between components written in C++ and components written in Java.

IPC — A set of techniques used in computing for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.
See Also inter-process communication.

M

MoCCA — Component based framework used by HARMAN as a foundation for software development.
See Also modular car computing architecture.

X

XML — A set of rules for encoding documents in machine-readable form. It is defined in the XML specification produced by the W3C, and several other related specifications, all gratis open standards.
See Also extensible markup language.