Design Document

Contents

Interface Details..... 7

Design Scope..... 8

Design Methodology..... 13

Design Notations..... 14

Design Considerations..... 15

Design overview..... 24

Architecture Design..... 25

DIL_Interface..... 25
IDIL_CAN..... 26
DIL_ICS_neoVI..... 27
DIL_ETAS_BOA..... 28
DIL_Stub_CAN..... 29
ConfigDialogDIL..... 30
Bus Statistics..... 31
Simulation Engine..... 33
Project Configuration..... 37
Logger..... 41
Message Window..... 45
Filter..... 47
Session Replay..... 47
Node Simulation..... 49
Frame Transmission..... 54
Signal Watch Window..... 54
Test Automation..... 56
Test Setup Editor Lib module..... 57
Test Suite Editor and Executor Lib..... 58
Test Setup Editor..... 59
Automation Server Interface..... 60

Interface Design..... 62

User Interface..... 62
Component Interface..... 62
Test Suite Editor and Executor Lib..... 62

Design Alternatives..... 64

Design Feasibility..... 65

Design Tools used.....66

Additional Hardware and Software required.....67

Test Strategy.....68

References..... 69

License Information

This is a free document: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this artifact. If not, see <*http://www.gnu.org/licenses/*>.

Acknowledgement

The BUSMASTER application suite evoluted from its incipient stage of monolithic single bus - single controller architecture towards the present stage of modular & reusable component based architecture, easily and seamlessly extendable to support multiple vehicle buses, exhibiting a rich set of carefully crafted feature list. Reaching of such a milestone was possible only by the valuable contributions from many dedicated and skilful professionals who were involved at different cycles of the development process defining the trajectories of the product's evolution, translating many innovative ideas into reality. The contributors so far are Mrs. Jalaja K.V, Mr. Amitesh Bharti, Mr. Amarnath Shastry, Mr. Ratnadip Choudhury, Mr. Pemmaiah B.D, Mr. Raja N, Mr. Rajesh Kumar R, Mr. Anish Kumar, Mr. Pradeep Kadoor, Mr. Arunkumar Karri and Mr. Venkatanarayana Makam.

This design document is the summation of all the relevant design documents, notes and an analysis report, produced and refined throughout the development process, and hence is an integrated version prepared to roll out the OSS version of BUSMASTER.

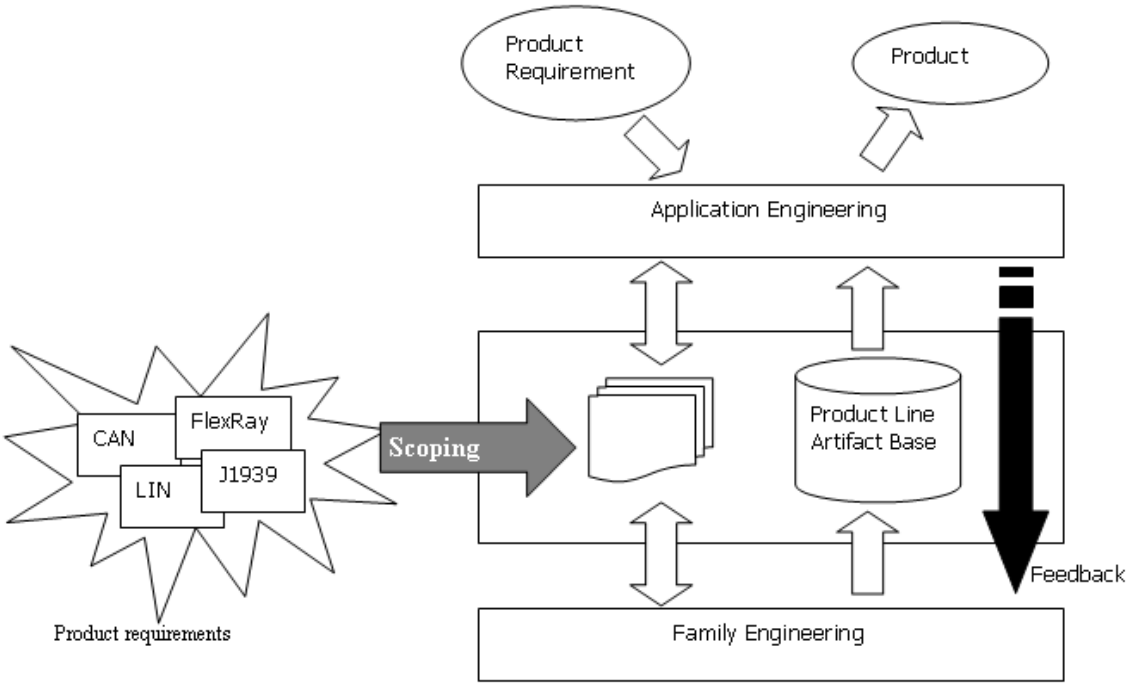
List of Abbreviations

Abbreviations, non-standard terms and acronyms that are used in this document are to be listed here.

API	Application Programming Interface
BOA	Basic Open Architecture
CAN	Controller Area Network
DIL	Driver Interface Library
FlexRay	A vehicle communication system developed by the FlexRay consortium
GUI	Graphic User Interface
HLD	High Level Design
J1939	A CAN based higher layer vehicle bus standard
LIN	Local Interconnect Network
OOD	Object Oriented Design
OSS	Open Source Software
RBEI	Robert Bosch Engineering and Business Solutions Limited
SEI	Software Engineering Institute
UI	User Interface

Interface Details

This document describes at length the various design aspects of BUSMASTER tool family which aims at simultaneous support of any vehicle bus standard.



For obvious reason the design approach primarily harps on the principle of large-scale reuse and consequently is broadly inspired by product line model to leverage on this said aspect. The approach conforms to the SEI definition of product line engineering which is “A set of systems sharing a common set of features that satisfy the specific needs of a particular market segment or mission and thus are developed from a common set of core assets in a prescribed way”. So outlining of the engineering activity may be presented by the following:

Design Scope

The modules mostly have a one-to-one mapping with the feature set the application suite. The feature list against the implementing modules is tabulated below:

SI. No.	Feature Name	Module Name	Remark
1	Project configuration saving and retrieval	ProjectConfiguration.dll	Used to store project configuration data in a binary file which also can accommodate multiple project information. If necessary, ODBC support can also be brought in; this is supported by its design.
2	Interface for a DIL of a bus	DIL_Interface.dll	DIL stands for Driver Interface Layer and is the interface to the bus. This abstracts out the driver details from the application thereby bringing in loose coupling. When the query for a particular bus interface from the client component to DIL_Interface.dll is successful, this library renders the appropriate interface.
3	CAN DIL ICS neoVI	CAN_ICS_neoVI.dll	CAN DIL interface to neoVI controllers from IntrepidCS
4	CAN DIL PEAK USB	CAN_PEAK_USB.dll	CAN DIL interface to PCAN controller from Peak GmbH
5	CAN DIL ETAS BOA	CAN_ETAS_BOA.dll	Interface to BOA from ETAS which is an abstraction to its CAN hardware interfaces. This completely decouples the client application from the underlying hardware.
6	CAN controller configuration	ConfigDialogsDIL.dll	Implements the controller configuration GUI. This makes it possible for BUSMASTER application to completely get decoupled from the underlying hardware support. Addition of a new hardware / controller for a bus means a updating in DIL_<bus>.dll, ConfigDialogs.dll and introduction

SI. No.	Feature Name	Module Name	Remark
			of <bus>_<new controller>.dll. Commercial feature related files & routines will be taken off.
7	CAN bus simulation	CAN_STUB.dll & BusEmulation.exe	The first one is CAN DIL interface to simulation engine. Second one is also known as simulation engine; this realizes a virtual bus in the workstation making it possible for any instance of BUSMASTER to act as a node and communicate via this communication channel. By design this should be able to simulate any bus hitherto known and unknown. Also, the same instance of server can cater to any number of buses with any number of nodes simultaneously.
8	Filtering	Filter.dll	Implements filtering configuration dialogues. Both a composite aggregate and generic. Commercial feature related files & routines will be taken off.
9	Node programming and simulation for CAN	NodeSimEx.dll	Realizes greater part of the node programming features. Accommodates both function editor and node executor modules. Function editor is generic and is configurable by its client on the fly to cater to various bus interface characteristics. Node executor - on the other hand contains individual bus specific slim sub-modules with majority of common modules / classes.
10	Logging of frame data	FrameProcessor.dll	Is utilized to log messages. When queried, this provides interface to various sub modules specific to individual buses. Since logging procedure is same for any bus, save the protocol specific frame information, the classes

SI. No.	Feature Name	Module Name	Remark
			all descend from a common class and employs common worker thread function. This results in saving a lot of implementation and testing effort. For example - adding J1939 support took 3.5 days of effort instead of the usual 2.5 weeks of effort, thereby saving 72%.
11	Play back a session	Replay.dll	Implements the replay module, covering both configuration and operation aspects. Tested so far only for CAN.
12	Signal watch - CAN	SignalWatch.dll	Implements signal watch windows. Dynamically configurable for any bus.
13	CAN Frame transmission - both in mono shot and cyclic modes	TxWindow.dll	Implements message transmission configuration UIs and helper modules. Implemented so far for CAN.
14	Signal graph	SigGrphWnd.dll & CGCtrl.ocx	**1.** Responsible for managing the signal value display in the signal graph activeX control. Configurable for any bus and so the bus-specific routines are responsibilities of the client module. **2.** This is the signal graph control where selected signal values are displayed as continuous line diagrams.
15	CAN message window	PSDI_CAN.dll	A helper module for CAN message window containing all the bus specific associated routines.
16	Test Automation	TestSetupEditorGUI.dll, TestSetupEditorLib.lib, TestSuiteExecutorGUI.dll & TestSuiteExecutorGUI.lib	**1.** This is the editor of a test setup file. **2.** This is the helper library exclusively for test setup editor **3.** This defines a test suite, executes the same and saves the

SI. No.	Feature Name	Module Name	Remark
			result in the desired format.
17	Database editor	BUSMASTER.exe	BUSMASTER application. Full decoupling is not yet fully achieved; this contains some generic classes, modules as well. Commercial feature related files & routines will be taken off.
18	Bus statistics	BUSMASTER.exe	Same as above
19	Automation server interface	BUSMASTER.exe	Same as above
20	Log file export	BUSMASTER.exe	Same as above
21	Trace window	BUSMASTER.exe	Same as above
22	Main GUI and controller & configuration for other modules	BUSMASTER.exe	Same as above
23	Helper libraries (not feature)	DataType.lib	Implements the data types / structures that are widely utilised by any module / component of BUSMASTER tool family. This was created with the idea to solve the greater part of the problem at hand by properly defining the data type. Commercial feature related files & routines will be taken off.
24	Helper libraries (not feature)	Utils.lib	Contains some utility services, both GUI and non-GUI, commonly soughted after by any module / component of BUSMASTER tool family. Commercial feature related files & routines will be taken off.
25	Helper libraries (not feature)	CommonClass.lib	Implements some utility classes commonly used. Commercial feature related files & routines will be taken off.
26	Wave pattern generator	SignalGeneration.dll, BUSMASTER.exe	This generates signals with values matching certain regular wave patterns like sinusoidal, triangular, spike etc. For CAN the first two has been implemented.

This design document addresses all the required different design aspects namely:

1. Component diagram
2. State machine diagram
3. Sequence diagram
4. Class diagram
5. Deployment diagram

So for every module the applicable design will be discussed on.

Design Methodology

Hybrid design methodology consisting of structured and object oriented has been used.

Design Notations

- For design, UML 2.0 notation is used
- For class nomenclature, the RBEI/ECM coding guideline has been followed.

Design Considerations

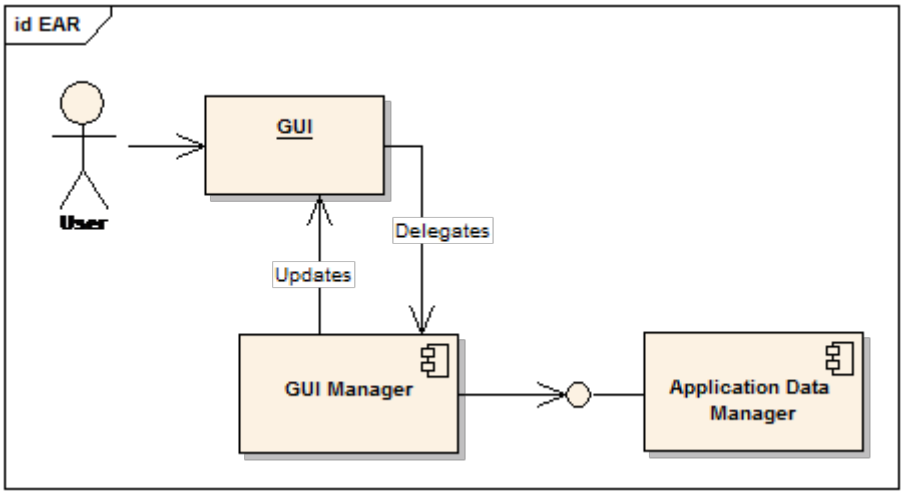
Here some the design requirements which are to be followed while carrying out development activities are described concisely. The BUSMASTER application design & architecture specification, following continuous associated improvement activities, evoluted from a vast monolithic architecture into a properly harmonised ensemble of loosely coupled reusable and efficient modules & components. As a natural and obvious consequence, certain patterns and structural approaches for development, albeit implicit, resulted. This document aims at capturing those steps and implied procedures.

The various aspects can be listed as:

1. Adequate loose coupling between UI and data: This application is GUI based for normal usage. However, usage as automation server is also possible.

   Towards this, a problem arises which is related to the complexity in harmonization of GUI with the application data (backend data).

   The user modifies the application data through the GUI which is a coarse visual reflection of the data. The handler modifies the data as per the user's choice and domain rules and then updates the GUI based on the handler or function data.



It is often missed that GUI is only one of the ways to initiate application data modification activity. Explicitness of the other one remains unconsidered most often. Therefore, the design and the very structuring of the sub modules & function list naturally don't support this aspect. The result is obvious – whenever such a need arises (e.g., exploitation of the feature from some other module while keeping the GUI on, automated testing etc), bugs with synchronization of GUI with the application data, unexpected crashes, duplication of codes etc creep in. This results in patch work, which further pervades into chaotic coding and many other undesirable consequences.

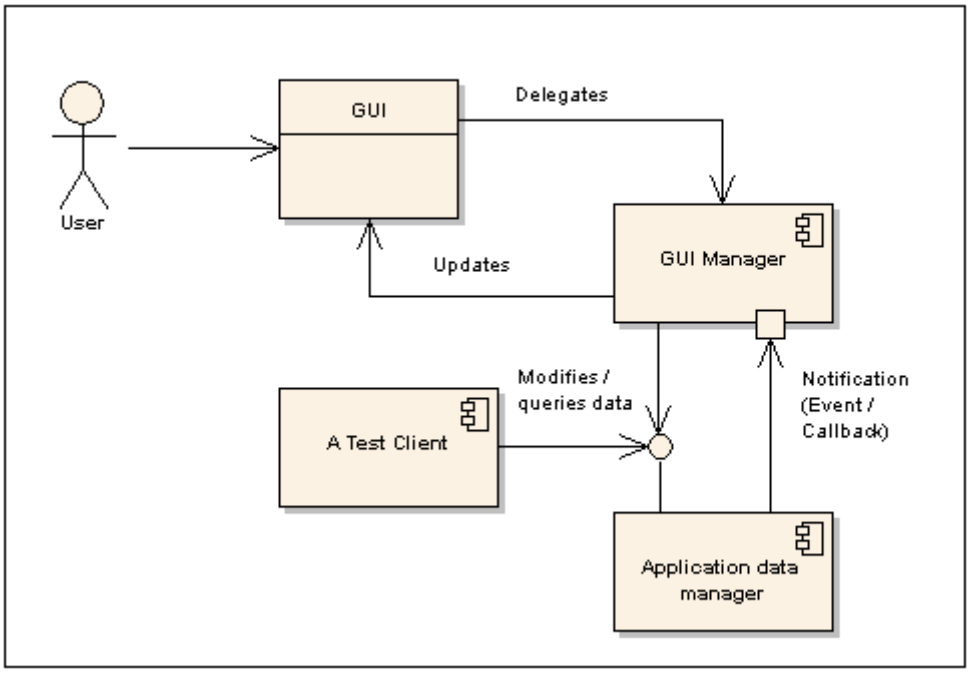
A sample code is presented below. The use case deals with 5 controls and takes the services of 5 utility API calls from the application data manager library.

So long as we use the GUI to run this particular use case, things will proceed as expected, in the right manner. However, when it is done through some other means instead, (e.g., from a tester module running the application in the server mode, running the use case from another sub module etc). Something unexpected happens most often. In case of the former, a crash in case the GUI code doesn't always validate the properness of the window or control handles or object pointers. In case of the later, either duplicate code and hence possibility of the GUI not appropriately reflecting the application data state. Add with it the numerous GUI conditions.

Very defensive coding may solve the problem. Nevertheless – it should always be kept in mind that that is hard to practice and also this can make the code rather ugly and unreadable. Hence, a design solution may be the most natural and therefore, best approach.

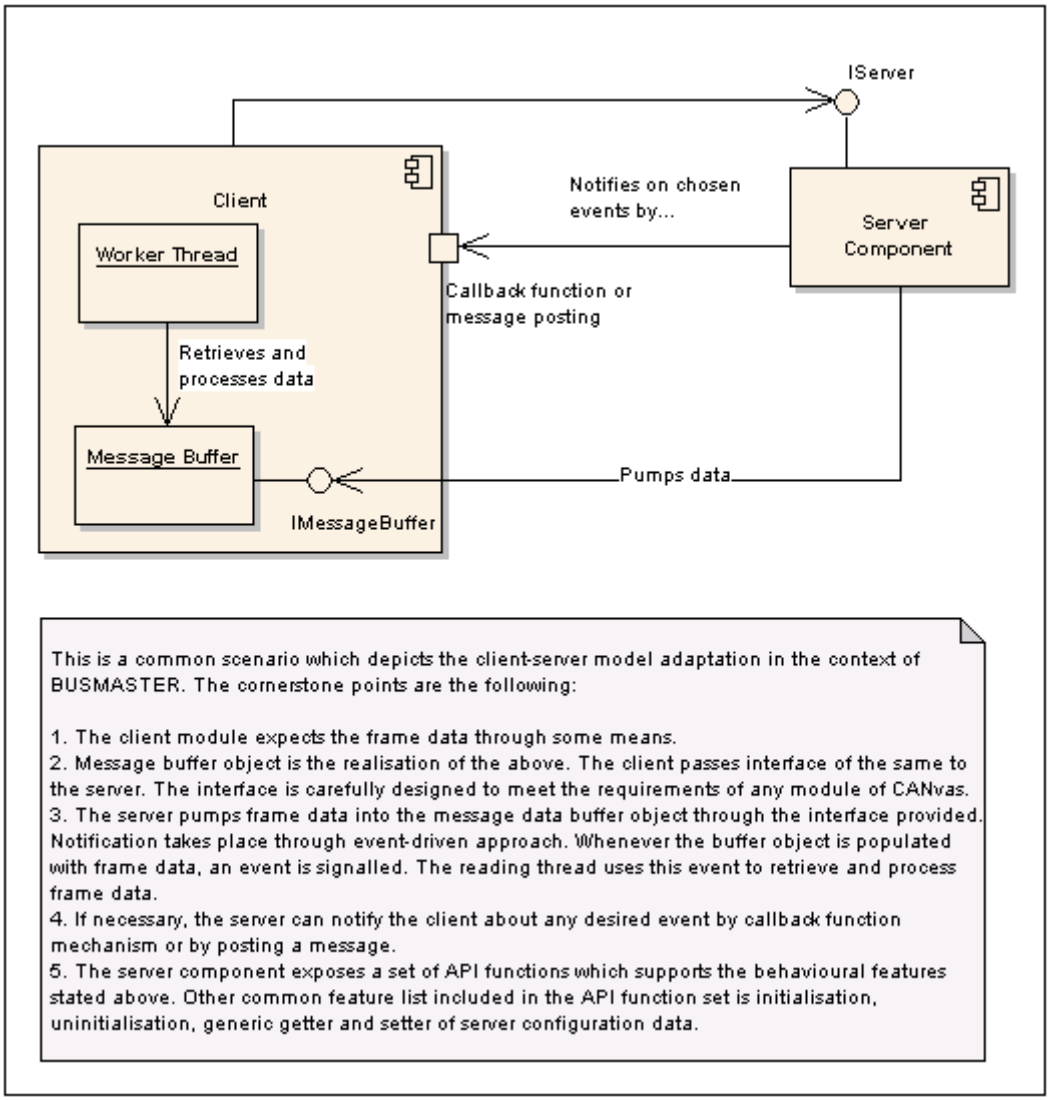
```
// Data_Manager.cpp

RET_TYPE Function1_Exported(ARG_TYPE);
RET_TYPE Function2_Exported(ARG_TYPE);
RET_TYPE Function3_Exported(ARG_TYPE);
RET_TYPE Function4_Exported(ARG_TYPE);
RET_TYPE Function5_Exported(ARG_TYPE);


// GUIClass.cpp
#include "Data_Manager_extern.h"

// Set of functions that operate on Control 1 upto 3
void GUIClass::Modify_Control1(ARG_TYPE)
{
// Modify control1 according to ARG_TYPE
}

void GUIClass::Modify_Control2(ARG_TYPE)
{
// Modify control1 according to ARG_TYPE
}

void GUIClass::Modify_Control3(ARG_TYPE)
{
// Modify control1 according to ARG_TYPE
}

void GUIClass::Handler(ARG_TYPE)  // The handler implements the logic
 to realize
{      // the desired functionality.
if (Function1_Exported(..) == SUCCESS)
{
Modify_Control3(..);

if (Function5_Exported(..) == SUCCESS)
{
if ( Function3_Exported(..) && Function2_Exported(..) )
{
// Code to update control 4
Modify_Control2(..);
}

if (Function4_Exported(..) == FAILURE)
{
Modify_Control1(..);
}
else
{
// Code to update control 5
}
}
}
}
```

The solution crafted is layering of responsibilities by introducing a two-way communication with the data source. Any change in the back-end data can be indicated to the existing clients implemented either in event notification or call-back function. So the back-end data or the application data manager should have a well-defined interface. A typical use case can be realized by the execution of a few routines in some order, with adequate parameters.



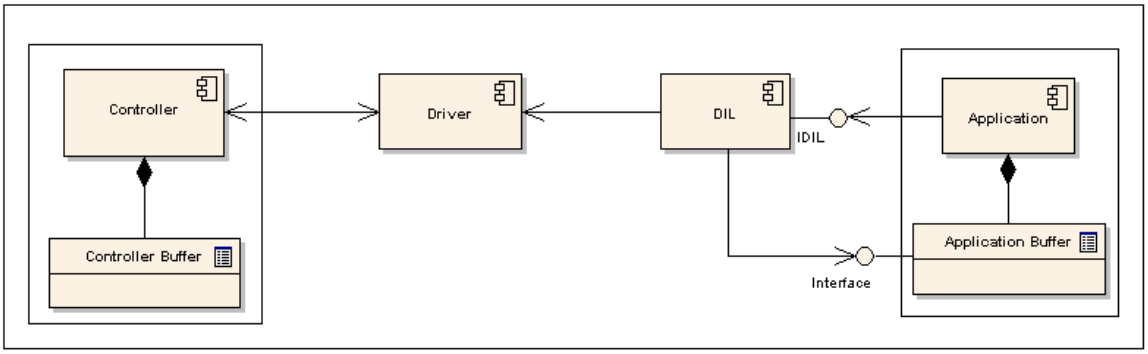
The solution concept is inspired on the observer design pattern and the model-view-controller software architecture.

2. Abstract data channel: Usage of BUSMASTER falls under two distinct categories namely, bus monitoring & analysis, and node simulation. Undercurrent of the first one is “retrieve data and represent the same in the desired fashion”. All the major functionalities like frame display, logging, data interpretation, signal watch and graph, replay etc have frame data processing as the kernel of their activities. That’s why it is of paramount importance to make the data retrieval procedure efficient and simple. The below diagram depicts such a schema ~



This calls for an optimised design for the message buffer classes which also qualifies the necessary performance criteria. Since frame data processing lies as the core of the activities, this issue is of paramount importance and even an iota of improvement in this direction can significantly contribute is performance elevation. The next point details on this.

3. Optimised data design: Reiterating the RS, this discussion continues with the disclaimer that no loss of frame data can't be guaranteed. Only the possibility of data loss can be brought down with efficient design of the data structure. In a non-real time system like Windows this can be achieved by using buffer(s) as temporary placeholder(s) of data as shown in the figure above. The controller in addition contains a buffer that provides the first level of buffering whereas the application buffer is the last one. Hence we have an ensemble of two buffers – one from the controller and the other from the application.



Hence the deciding factors can be the following:

1. Size of the controller buffer
2. Controller buffer management
3. Size of the application buffer
4. Application buffer management
5. Retrieval process of frame data from the driver.

'A' and 'B' are out of the purview of our discussion on analysis and strategy.

Optimum size calculation of the application buffer depends on the following factors ~

1. A.Current bit rate / throughput per second. It is finally the amount of data per second / millisecond that matters and hence the upper limit of total data at any moment.
2. B.Space management of each frame entry. Since the payload / data segment length of frames of buses like FlexRay or J1939 varies within a known range, a buffer of maximum allocable size can cater to a frame of any allowable length. In another approach, only the space of the actual size may be used – given we know the current bit rate. The former approach may be called method of fixed-size entry whereas the later one may be termed as method of variable-size entry.

Formulation for method of variable-size entry

Let rt bps be the data rate at any moment t. If T seconds is the total time for which the bus data are to be retained, then the total amount of space can be expressed as -

$S = \Sigma rt$, where $1 <= t <= T$.

Assuming R be the average data rate,

$S = R * T$

Let C be the controller buffer size and M be the application buffer size. Hence, the following inequality must hold good -

$R * T < C + M$

Considering the bus to be FlexRay, for each channel,

Rmax = 10 Mbps

Rmax = 10 * 2^17 bps

Counting both the channels, the total data rate is 10 * 218 Bps

Calculation of the optimized value of application buffer size depends on the realistic value of R and correct value of C.

Assuming maximum data rate and neglecting C, we find for FlexRay ~

$M > 2.5 * T$ MB

The payload segment of a frame varies from 0 up to 127 WORDs. Hence the above formula is useful when the buffer for a frame is not assumed to be of fixed size and hence follows the implementation.

Similarly, for a CAN controller with a single channel -

Rmax = 1 Mbps

Rmax = 2^17 Bps

Again assuming maximum data rate and neglecting C, we find for CAN ~

M > 1^27 * T kB

Formulation for method of fixed-size entry

The first use case cited is for FlexRay bus.

If Fcurr is the size in bytes of the current frame, then

Fcurr = 8 + 2 * W, where 0 <= W <= 127 and W is the number of WORDs in the payload segment.

If N is the number of Fcurr frames, then the following equation will hold good,

R * T = N * Fcurr

N = (R * T) / (8 + 2 * W)

For evident reason, space for every entry is allocated to accommodate frame of maximum size i.e., Fmax = 262 bytes.

Therefore, total amount of space needed is,

M = N * Fmax

M = (R * T * 262) / (8 + 2 * W)

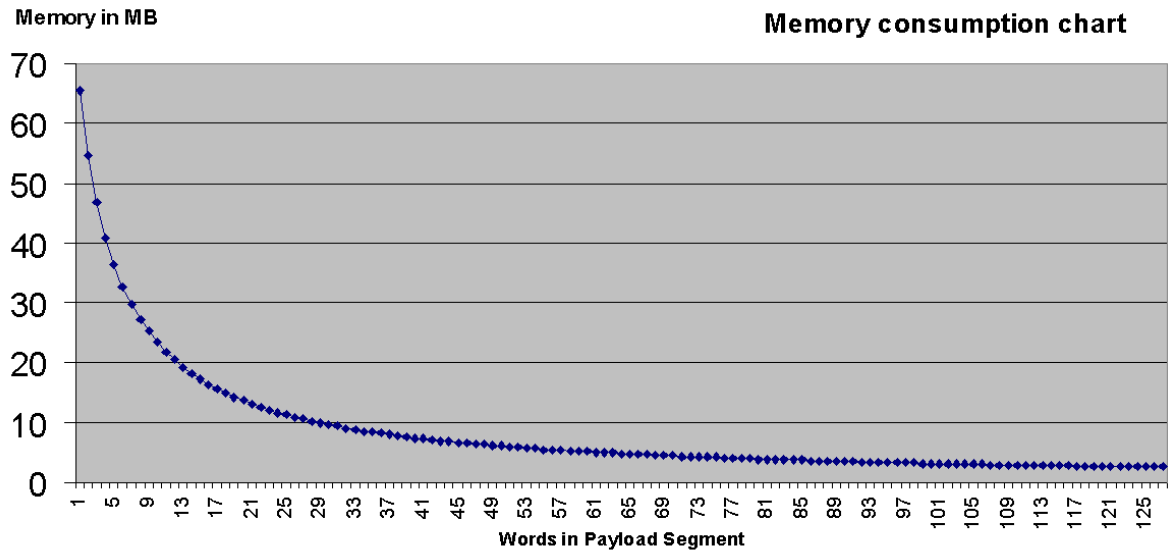
Assuming maximum data rate accounting both the channels,

M = (2 * Rmax * T * 262) / (8 + 2 * W)

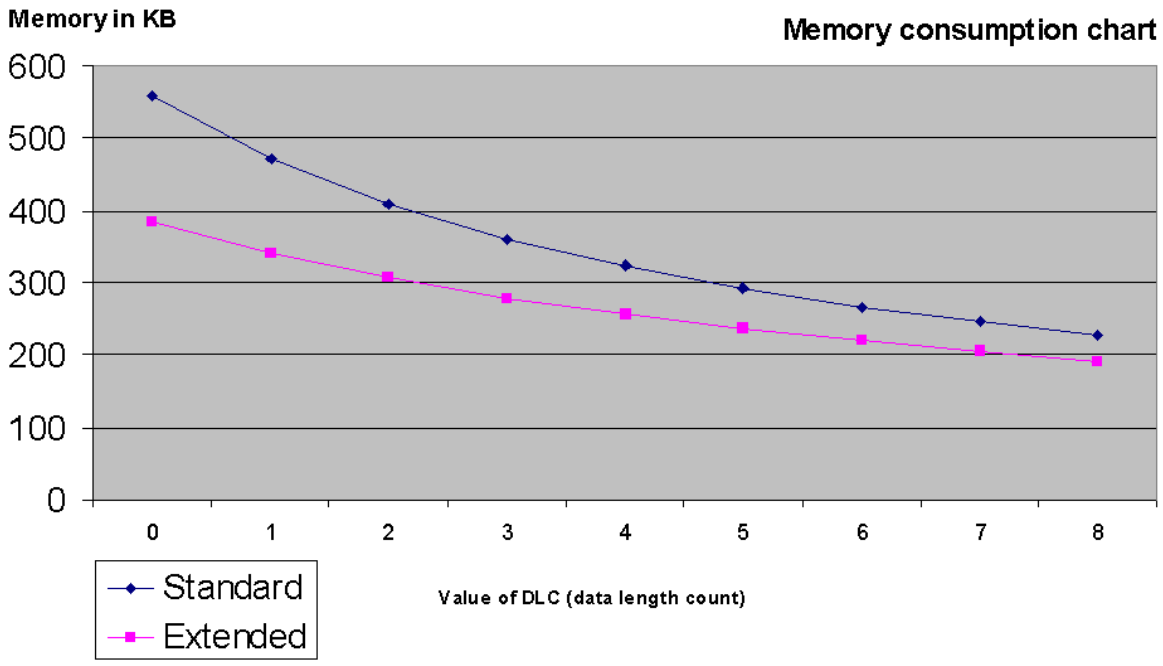
M = (10 * 218 * T * 262) / (8 + 2 * W), in MB

Clearly, M and W are inversely proportional.

The following graph shows the amount of memory to be allocated for different payload segment length when the data rate is 10 Mb/s for each channel and the data need to be contained for a second.



Similarly, for CAN also the same calculation may be carried out in which both normal and extended frames are considered. Maximum baud rate is assumed and for each data length code value ranging between 0 and 8, the number of entries in fixed-size entry buffer is calculated out. The consolidated result is represented with a line diagram. Below is shown the application buffer size nuance chart:



DLC	Frames per second	Memory (KB)	DLC	Frames per second	Memory (KB)	Data Rate
0	23831.27273	558.5454545	0	16384	384	1048576
1	20164.92308	472.6153846	1	14563.55556	341.3333	Duration
2	17476.26667	409.6	2	13107.2	307.2	1
3	15420.23529	361.4117647	3	11915.63636	279.2727	Static section (standard CAN)
4	13797.05263	323.3684211	4	10922.66667	256	44
5	12483.04762	292.5714286	5	10082.46154	236.3077	Frame length
6	11397.56522	267.1304348	6	9362.285714	219.4286	192
7	10485.76	245.76	7	8738.133333	204.8	Static section (extended CAN)
8	9709.037037	227.5555556	8	8192	192	64

Type	Field	Length (Bits) [Min]	Length (Bits) [Max]	Purpose
Base Frame Format	Start-of-frame	1	1	Denotes the start of frame transmission
Base Frame Format	Identifier	11	11	A (unique) identifier for the data
Base Frame Format	Remote transmission request (RTR)	1	1	Dominant (0) (see Remote Frame below)
Base Frame Format	Identifier extension bit (IDE)	1	1	Must be dominant (0) Optional

Type	Field	Length (Bits) [Min]	Length (Bits) [Max]	Purpose
Base Frame Format	Reserved Bit (r0)	1	1	Reserved bit (it must be set to dominant (0), but accepted as either dominant or recessive)
Base Frame Format	Data Length Code (DLC)	4	4	Number of bytes of data (0-8 bytes)
Base Frame Format	Data field	0	64	Data to be transmitted (length dictated by DLC field)
Base Frame Format	CRC	15	15	Cyclic Redundancy Check
Base Frame Format	CRC Delimiter	1	1	Must be recessive (1)
Base Frame Format	ACK slot	1	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
Base Frame Format	ACK delimiter	1	1	Must be recessive (1)
Base Frame Format	End-of-frame (EOF)	7	7	Must be recessive (1)
Extended Frame Format	Start-of-frame	1	1	Denotes the start of frame transmission
Extended Frame Format	Identifier A	11	11	First part of the (unique) identifier for the data
Extended Frame Format	Substitute remote request (SRR)	1	1	Must be recessive (1) Optional
Extended Frame Format	Identifier extension bit (IDE)	1	1	Must be recessive (1) Optional
Extended Frame Format	Identifier B	18	18	Second part of the (unique) identifier for the data
Extended Frame Format	Remote transmission request (RTR)	1	1	Must be Dominant (0)
Extended Frame Format	Reserved Bit (r0, r1)	2	2	Reserved bit (it must be set to dominant (0), but accepted as either dominant or recessive)
Extended Frame Format	Data Length Code (DLC)	4	4	Number of bytes of data (0-8 bytes)

Type	Field	Length (Bits) [Min]	Length (Bits) [Max]	Purpose
Extended Frame Format	Data field	0	64	Data to be transmitted (length dictated by DLC field)
Extended Frame Format	CRC	15	15	Cyclic Redundancy Check
Extended Frame Format	CRC Delimiter	1	1	Must be recessive (1)
Extended Frame Format	ACK slot	1	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
Extended Frame Format	ACK delimiter	1	1	Must be recessive (1)
Extended Frame Format	End-of-frame (EOF)	7	7	Must be recessive (1)

The relative advantages and disadvantages of the two methods are tabulated below:

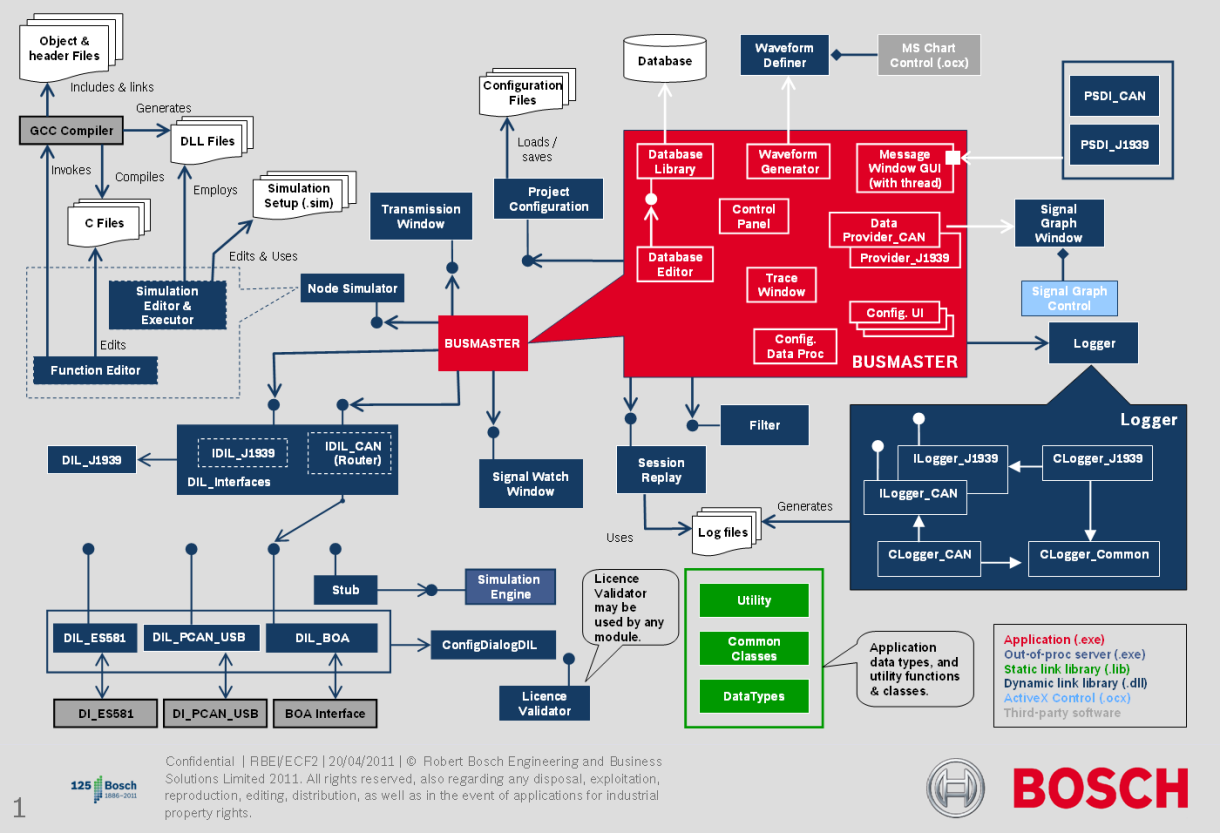
	Fixed-size entry	Variable-size entry
Resource usage	Variable; payload length of the frame is the deciding factor. Hence empirical data analysis is one of the means which is not guaranteed	Fixed; only the maximum bit rate is the deciding factor.
Easiness of implementation	Easier. There is no need to think on room for an entry.	Quite complex.

Variable-size entry method guarantees the most optimum usage of memory and addressing of any situation and hence real sturdiness. On the other hand, fixed-size entry method has a better performance. However, the difference in performance is not of a crucial magnitude. Therefore, choice of method is subject to the bus standard. For example – for CAN method of fixed-size entry is used whereas for J1939 the other one is used.

The application buffer class is the best agency to provide the information of buffer overrun. The ‘Write(…)’ method can check if the slot to write the new frame data is free. A new method can be exposed or an event can be fired to notify its client in case such occurrence. In fixed-size entry method it is also possible to keep count of the number of frames missed whereas in variable-size entry method it is not possible.

Design overview

The below diagram provides an overview of BUSMASTER application suite. This is a hybrid diagram combining interaction, component, and deployment views. Discussion under the architecture design section is based on this and the module list table.

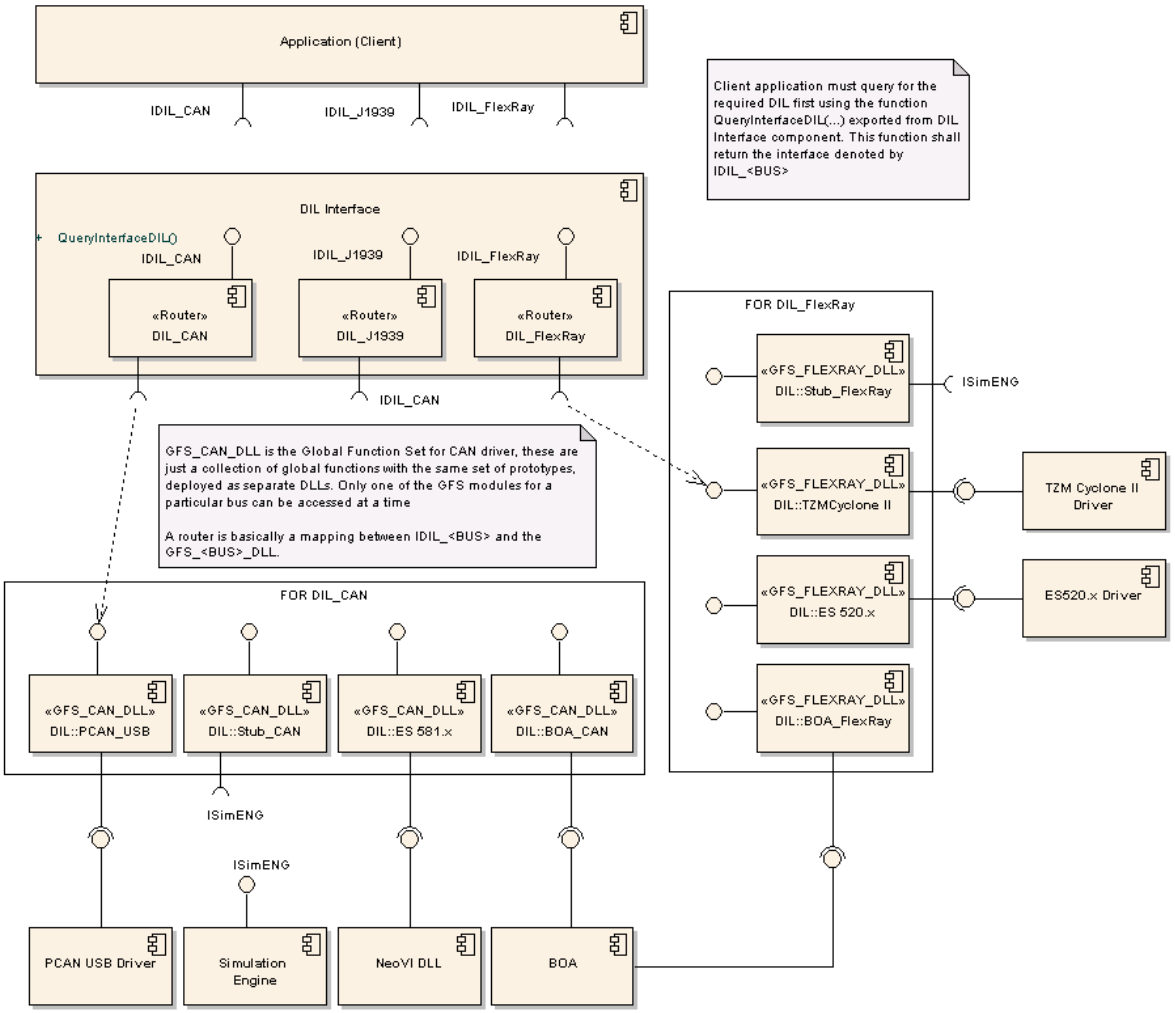


Architecture Design

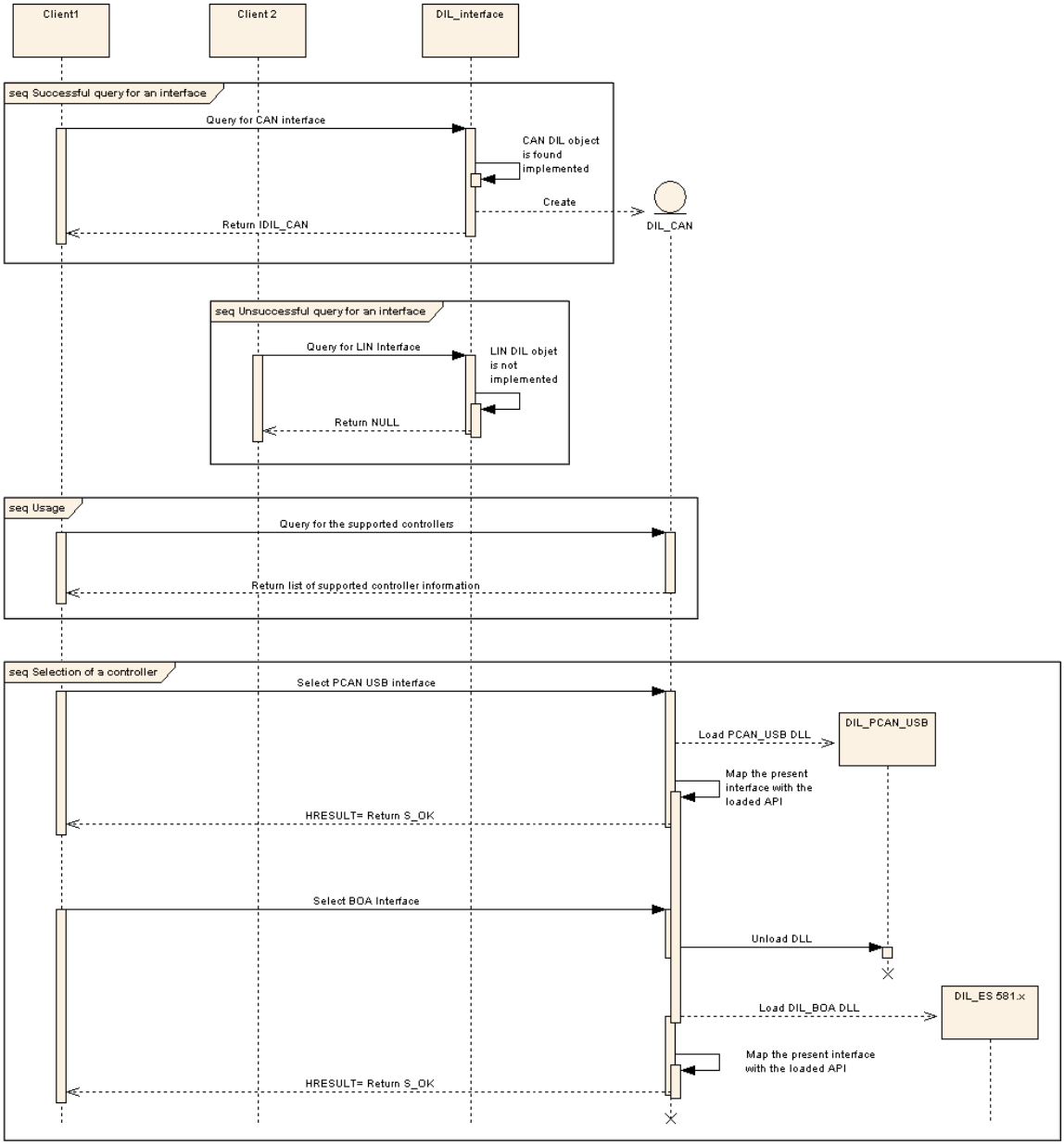
This section details about the individual logical entities covering the necessary views and interface details.

DIL_Interface

This module acts as the interfacing entity between the application and the expected DIL for the intended bus. This is the deliverer or interface manager of the DIL interface required by the application or the client. The procedure starts with the client's querying for the DIL and if it's available, DIL_Interface returns the interface pointer (IDIL_<Bus>) to the client. To get hold of a controller interface or even the simulation engine, a query must be made using the interface pointer. The whole arrangement is presented by the below component diagram covering support of CAN, J1939 and FlexRay with the support of a few controllers:

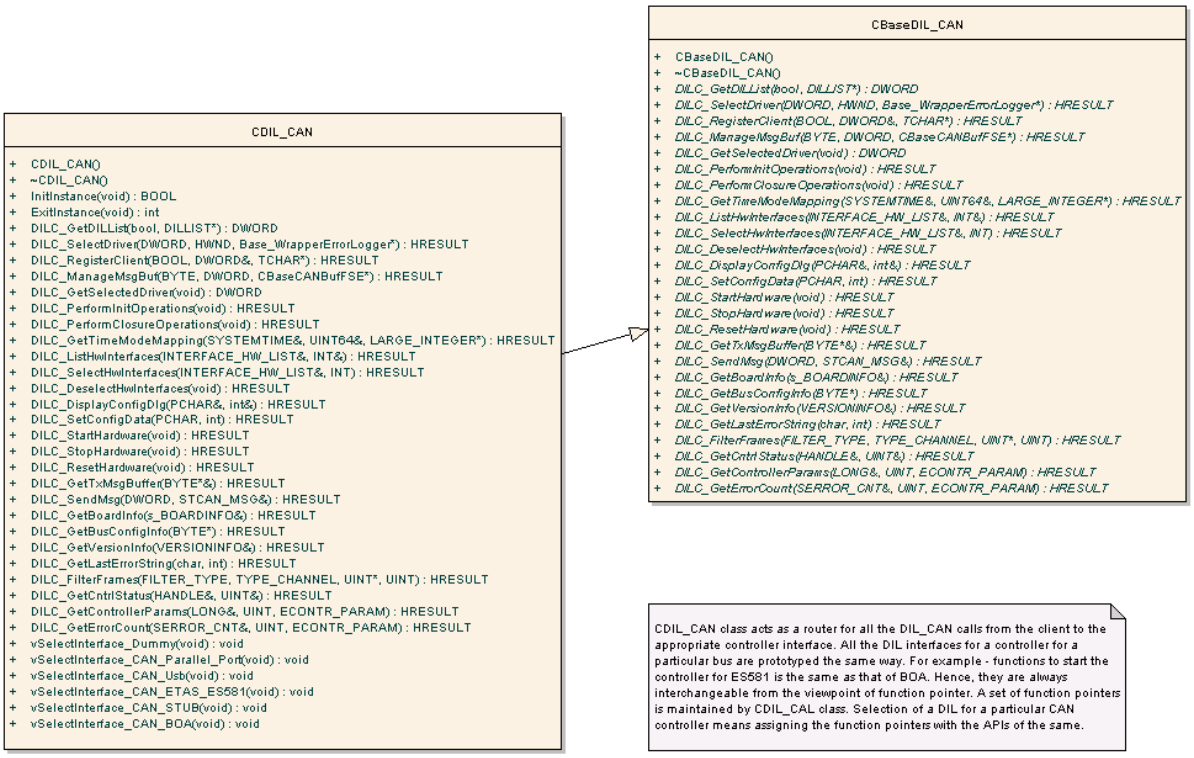


The calling sequence of query is illustrated by the following sequence diagram:



IDIL_CAN

This is the DIL interface of CAN. The class CDIL_CAN acts as the router, to maintain dynamic mapping between the interface and the target DIL function of the intended controller. The below class diagram provides a glimpse on that procedure.



Support of any controller needs writing of a socket for the same. This is implemented as a DLL and obviously the exported functions have the same prototype with the of the other controllers.

DIL_ICS_neoVI

This is the interface to IntrepidCS neoVI interfaces. Below goes the interface description:

```
// Feature function set

HRESULT CAN_BOA_PerformInitOperations(void);
HRESULT CAN_BOA_PerformClosureOperations(void);
HRESULT CAN_BOA_GetTimeModeMapping(SYSTEMTIME& CurrSysTime, UINT64&
TimeStamp, LARGE_INTEGER* QueryTickCount = NULL);
HRESULT CAN_BOA_ListHwInterfaces(INTERFACE_HW_LIST& sSelHwInterface, INT& nCount);
HRESULT CAN_BOA_SelectHwInterface(const INTERFACE_HW_LIST& sSelHwInterface, INT
nCount);
HRESULT CAN_BOA_DeselectHwInterface(void);
HRESULT CAN_BOA_DisplayConfigDlg(PCHAR& InitData, int& Length);
HRESULT CAN_BOA_SetConfigData(PCHAR pInitData, int Length);
HRESULT CAN_BOA_StartHardware(void);
HRESULT CAN_BOA_StopHardware(void);
HRESULT CAN_BOA_ResetHardware(void);
HRESULT CAN_BOA_GetCurrStatus(s_STATUSMSG& StatusData);
HRESULT CAN_BOA_GetTxMsgBuffer(BYTE*& pouFlxTxMsgBuffer);
HRESULT CAN_BOA_SendMsg(DWORD dwClientID, const STCAN_MSG& sCanTxMsg);
HRESULT CAN_BOA_GetBoardInfo(s_BOARDINFO& BoardInfo);
HRESULT CAN_BOA_GetBusConfigInfo(BYTE* BusInfo);
HRESULT CAN_BOA_GetVersionInfo(VERSIONINFO& sVerInfo);
HRESULT CAN_BOA_GetLastErrorString(CHAR* acErrorStr, int nLength);
HRESULT CAN_BOA_FilterFrames(FILTER_TYPE FilterType, TYPE_CHANNEL Channel, UINT*
punMsgIds, UINT nLength);
HRESULT CAN_BOA_GetControllerParams(LONG& lParam, UINT nChannel, ECONTR_PARAM
eContrParam);
HRESULT CAN_BOA_GetErrorCount(SERROR_CNT& sErrorCnt, UINT nChannel,
ECONTR_PARAM eContrParam);

// Infrastructural function set

HRESULT CAN_BOA_LoadDriverLibrary(void);
HRESULT CAN_BOA_SetAppParams(HWND hWndOwner, Base_WrapperErrorLogger* pILog);
HRESULT CAN_BOA_UnloadDriverLibrary(void);
HRESULT CAN_BOA_ManageMsgBuf(BYTE bAction, DWORD ClientID, CBaseCANBufFSE*
pBufObj);
HRESULT CAN_BOA_RegisterClient(BOOL bRegister, DWORD& ClientID, TCHAR*
pacClientName);
HRESULT CAN_BOA_GetCntrlStatus(const HANDLE& hEvent, UINT& unCntrlStatus);
```

DIL_ETAS_BOA

This is the interface to BOA interface. Below goes the interface description:

```
// Feature function set

HRESULT CAN_BOA_PerformInitOperations(void);
HRESULT CAN_BOA_PerformClosureOperations(void);
HRESULT CAN_BOA_GetTimeModeMapping(SYSTEMTIME& CurrSysTime, UINT64&
TimeStamp, LARGE_INTEGER* QueryTickCount = NULL);
HRESULT CAN_BOA_ListHwInterfaces(INTERFACE_HW_LIST& sSelHwInterface, INT& nCount);
HRESULT CAN_BOA_SelectHwInterface(const INTERFACE_HW_LIST& sSelHwInterface, INT
nCount);
HRESULT CAN_BOA_DeselectHwInterface(void);
HRESULT CAN_BOA_DisplayConfigDlg(PCHAR& InitData, int& Length);
HRESULT CAN_BOA_SetConfigData(PCHAR pInitData, int Length);
HRESULT CAN_BOA_StartHardware(void);
HRESULT CAN_BOA_StopHardware(void);
HRESULT CAN_BOA_ResetHardware(void);
HRESULT CAN_BOA_GetCurrStatus(s_STATUSMSG& StatusData);
HRESULT CAN_BOA_GetTxMsgBuffer(BYTE*& pouFlxTxMsgBuffer);
HRESULT CAN_BOA_SendMsg(DWORD dwClientID, const STCAN_MSG& sCanTxMsg);
HRESULT CAN_BOA_GetBoardInfo(s_BOARDINFO& BoardInfo);
HRESULT CAN_BOA_GetBusConfigInfo(BYTE* BusInfo);
HRESULT CAN_BOA_GetVersionInfo(VERSIONINFO& sVerInfo);
HRESULT CAN_BOA_GetLastErrorString(CHAR* acErrorStr, int nLength);
HRESULT CAN_BOA_FilterFrames(FILTER_TYPE FilterType, TYPE_CHANNEL Channel, UINT*
punMsgIds, UINT nLength);
HRESULT CAN_BOA_GetControllerParams(LONG& lParam, UINT nChannel, ECONTR_PARAM
eContrParam);
HRESULT CAN_BOA_GetErrorCount(SERROR_CNT& sErrorCnt, UINT nChannel,
ECONTR_PARAM eContrParam);

// Infrastructural function set

HRESULT CAN_BOA_LoadDriverLibrary(void);
HRESULT CAN_BOA_SetAppParams(HWND hWndOwner, Base_WrapperErrorLogger* pILog);
HRESULT CAN_BOA_UnloadDriverLibrary(void);
HRESULT CAN_BOA_ManageMsgBuf(BYTE bAction, DWORD ClientID, CBaseCANBufFSE*
pBufObj);
HRESULT CAN_BOA_RegisterClient(BOOL bRegister, DWORD& ClientID, TCHAR*
pacClientName);
HRESULT CAN_BOA_GetCntrlStatus(const HANDLE& hEvent, UINT& unCntrlStatus);
```

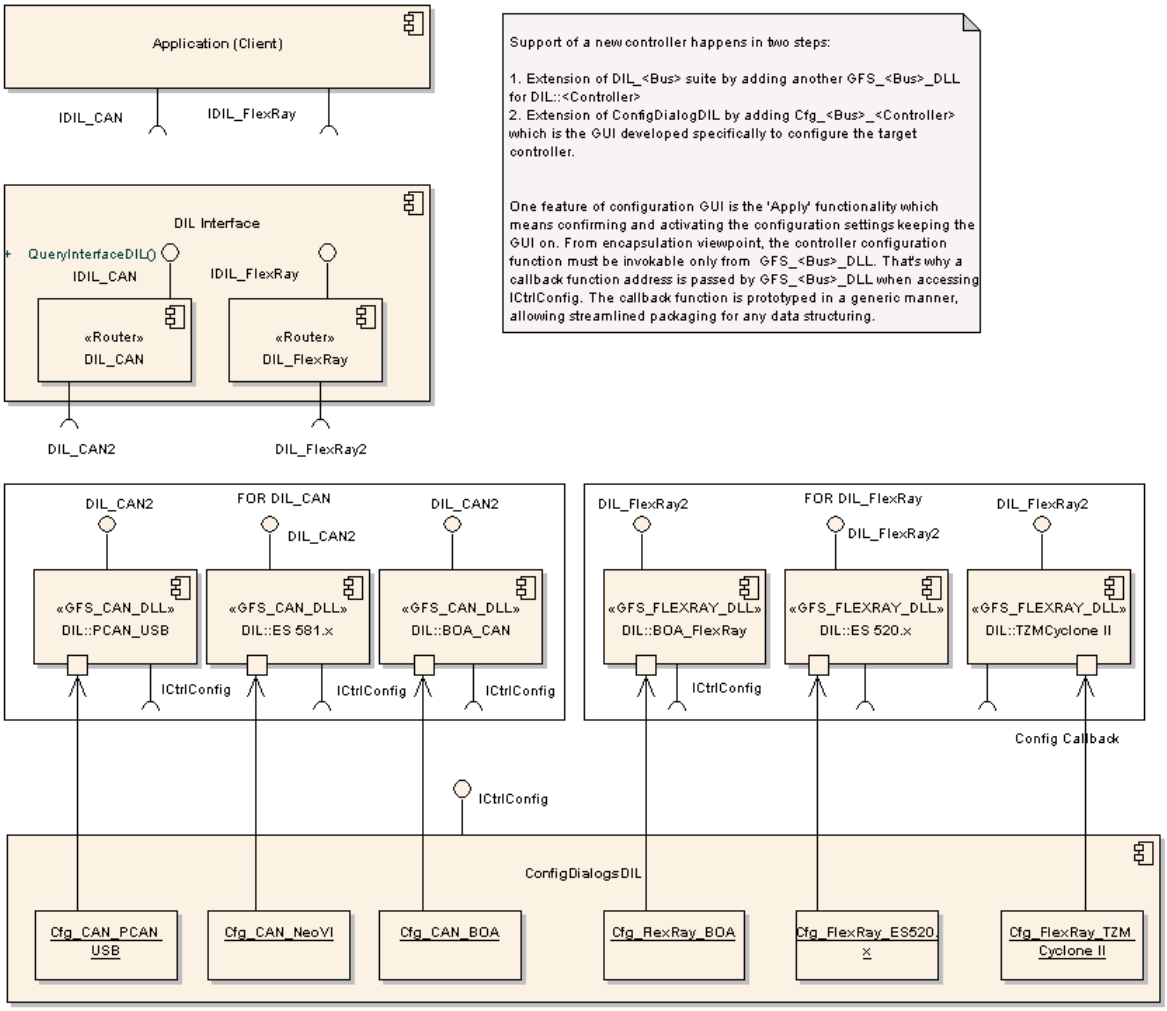
DIL_Stub_CAN

This is the interface to simulation engine interface. Below goes the interface description:

```
// Feature function set

HRESULT CAN_STUB_PerformInitOperations(void);
HRESULT CAN_STUB_PerformClosureOperations(void);
HRESULT CAN_STUB_GetTimeModeMapping(SYSTEMTIME& CurrSysTime, UINT64&
TimeStamp, LARGE_INTEGER* QueryTickCount = NULL);
HRESULT CAN_STUB_ListHwInterfaces(INTERFACE_HW_LIST& sSelHwInterface, INT&
nCount);
HRESULT CAN_STUB_SelectHwInterface(const INTERFACE_HW_LIST& sSelHwInterface, INT
nCount);
HRESULT CAN_STUB_DeselectHwInterface(void);
HRESULT CAN_STUB_DisplayConfigDlg(PCHAR& InitData, int& Length);
HRESULT CAN_STUB_SetConfigData(PCHAR pInitData, int Length);
HRESULT CAN_STUB_StartHardware(void);
HRESULT CAN_STUB_StopHardware(void);
HRESULT CAN_STUB_ResetHardware(void);
HRESULT CAN_STUB_GetCurrStatus(s_STATUSMSG& StatusData);
HRESULT CAN_STUB_GetTxMsgBuffer(BYTE*& pouFlxTxMsgBuffer);
HRESULT CAN_STUB_SendMsg(DWORD dwClientID, const STCAN_MSG& sCanTxMsg);
HRESULT CAN_STUB_GetBoardInfo(s_BOARDINFO& BoardInfo);
HRESULT CAN_STUB_GetBusConfigInfo(BYTE* BusInfo);
HRESULT CAN_STUB_GetVersionInfo(VERSIONINFO& sVerInfo);
HRESULT CAN_STUB_GetLastErrorString(CHAR* acErrorStr, int nLength);
HRESULT CAN_STUB_FilterFrames(FILTER_TYPE FilterType, TYPE_CHANNEL Channel, UINT*
punMsgIds, UINT nLength);
HRESULT CAN_STUB_GetControllerParams(LONG& lParam, UINT nChannel, ECONTR_PARAM
eContrParam);
HRESULT CAN_STUB_GetErrorCount(SERROR_CNT& sErrorCnt, UINT nChannel,
ECONTR_PARAM eContrParam);

// Infrastructural function set

HRESULT CAN_STUB_LoadDriverLibrary(void);
HRESULT CAN_STUB_SetAppParams(HWND hWndOwner, Base_WrapperErrorLogger* pILog);
HRESULT CAN_STUB_UnloadDriverLibrary(void);
HRESULT CAN_STUB_ManageMsgBuf(BYTE bAction, DWORD ClientID, CBaseCANBufFSE*
pBufObj);
HRESULT CAN_STUB_RegisterClient(BOOL bRegister, DWORD& ClientID, TCHAR*
pacClientName);
HRESULT CAN_STUB_GetCntrlStatus(const HANDLE& hEvent, UINT& unCntrlStatus);
```

ConfigDialogDIL

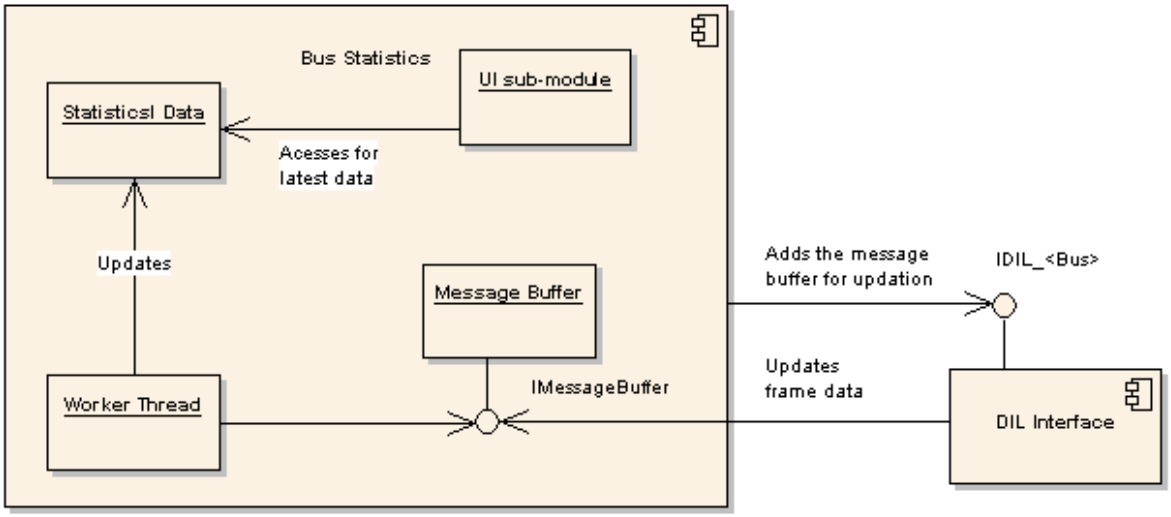
An overview of this module highlighting upon its constitutional and runtime behaviour is shown below:

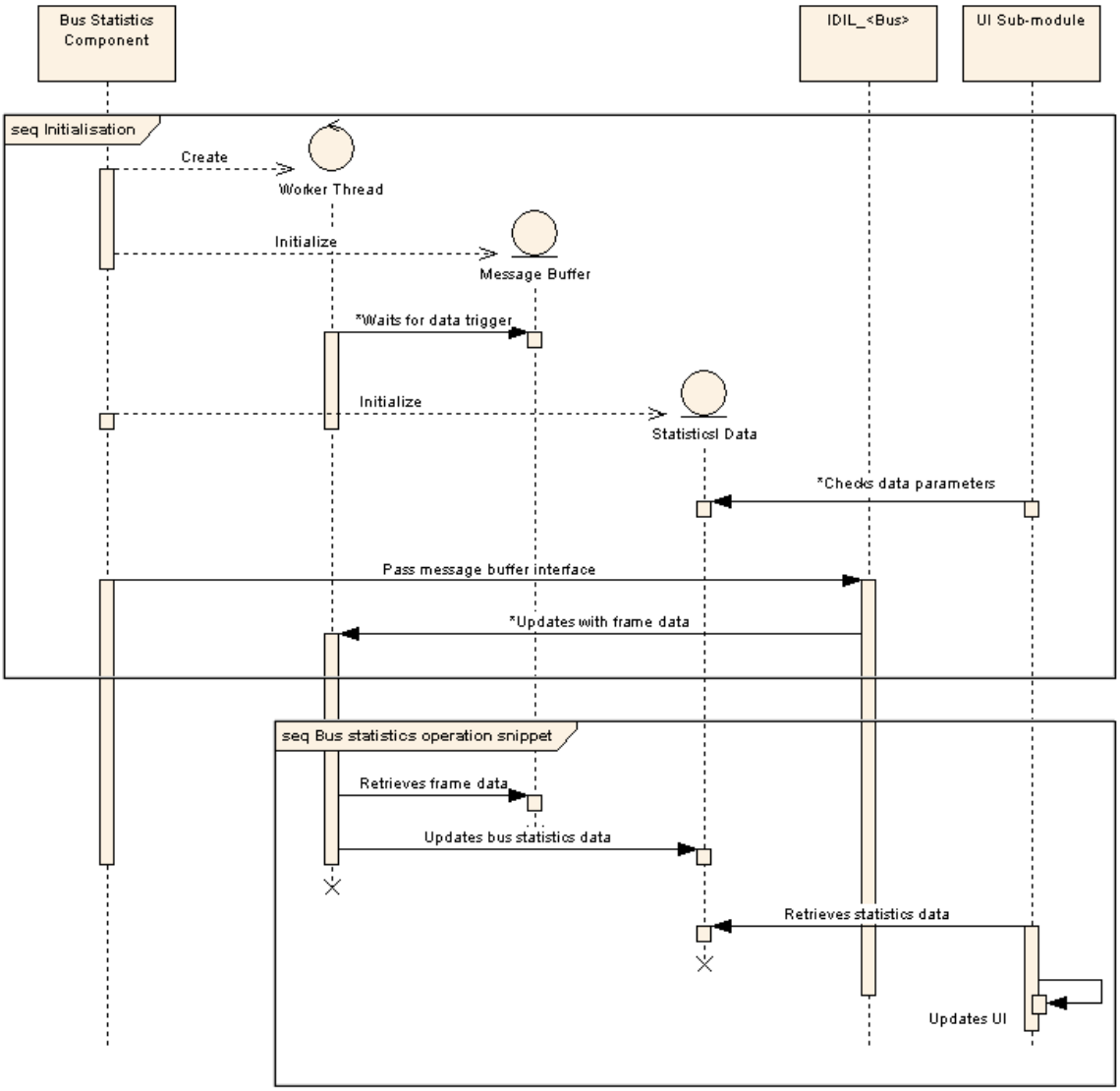


Bus Statistics

Like the other monitoring and analyzing modules, this Bus Statistics module implementation too follows the data channel abstraction procedure as already explained. This means – it owns a data buffer and uses it to register to the DIL as a data channel under the same client ID of its container – the application.

DIL ensures the buffer gets updated with the bus traffic. A worker thread maintained by bus statistics module waits for the notification event of the buffer and updates a set of bus statistics data parameters. The UI sub-module accesses the statistical data parameters at frequency according to the refresh rate set, and refreshes the required controls accordingly. This is depicted by the following component diagram:

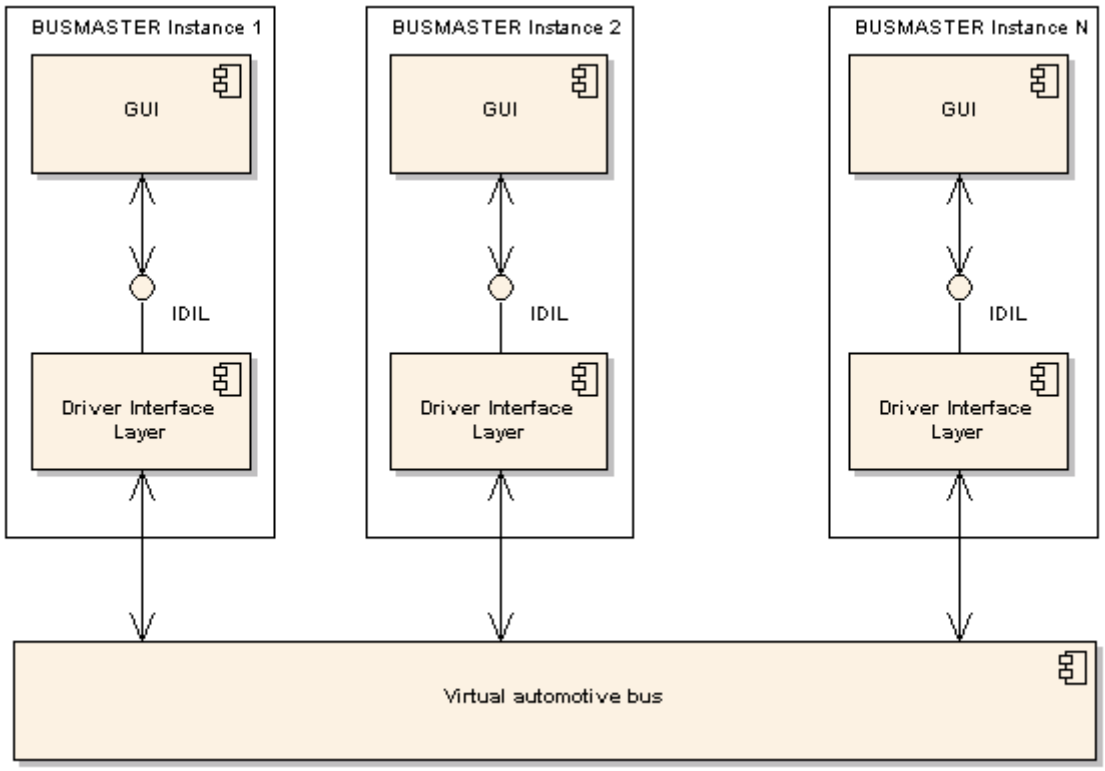

Bus Statistics
UI sub-module
StatisticsI Data
Acesses for
latest data
Updates
Message Buffer
Adds the message
buffer for updation
IDIL_<Bus>
Updates
frame data
IMessageBuffer
Worker Thread
DIL Interface


Bus Statistics
Component
IDIL_<Bus>
UI Sub-module
seq Initialisation
Create
Worker Thread
Initialize
Message Buffer
*Waits for data trigger
Initialize
StatisticsI Data
*Checks data parameters
Pass message buffer interface
*Updates with frame data
seq Bus statistics operation snippet
Retrieves frame data
Updates bus statistics data
Retrieves statistics data
Updates UI

Below is provided the class diagram:

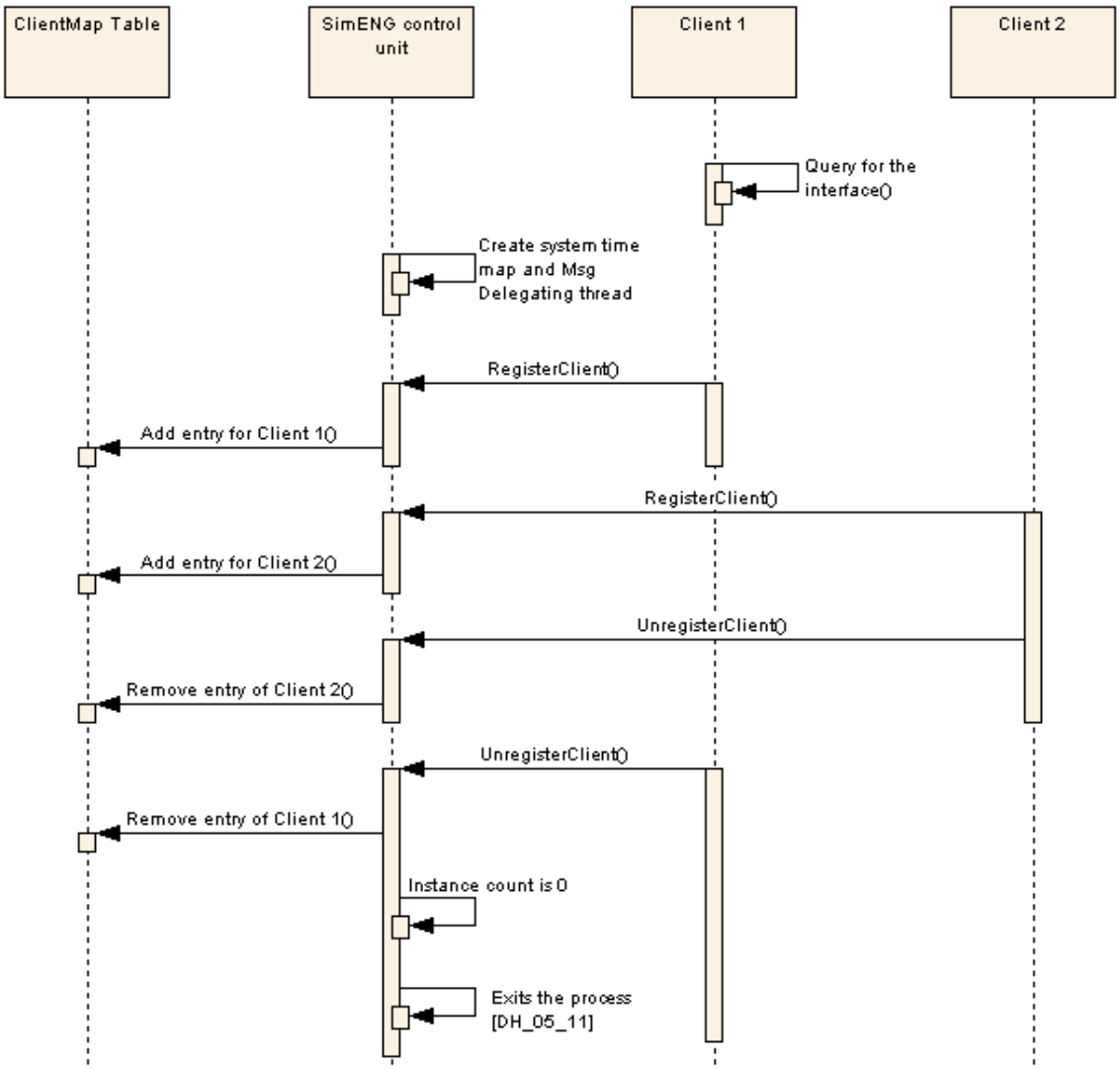
Simulation Engine

We start with a brief recapitulation on the overview of simulation engine.

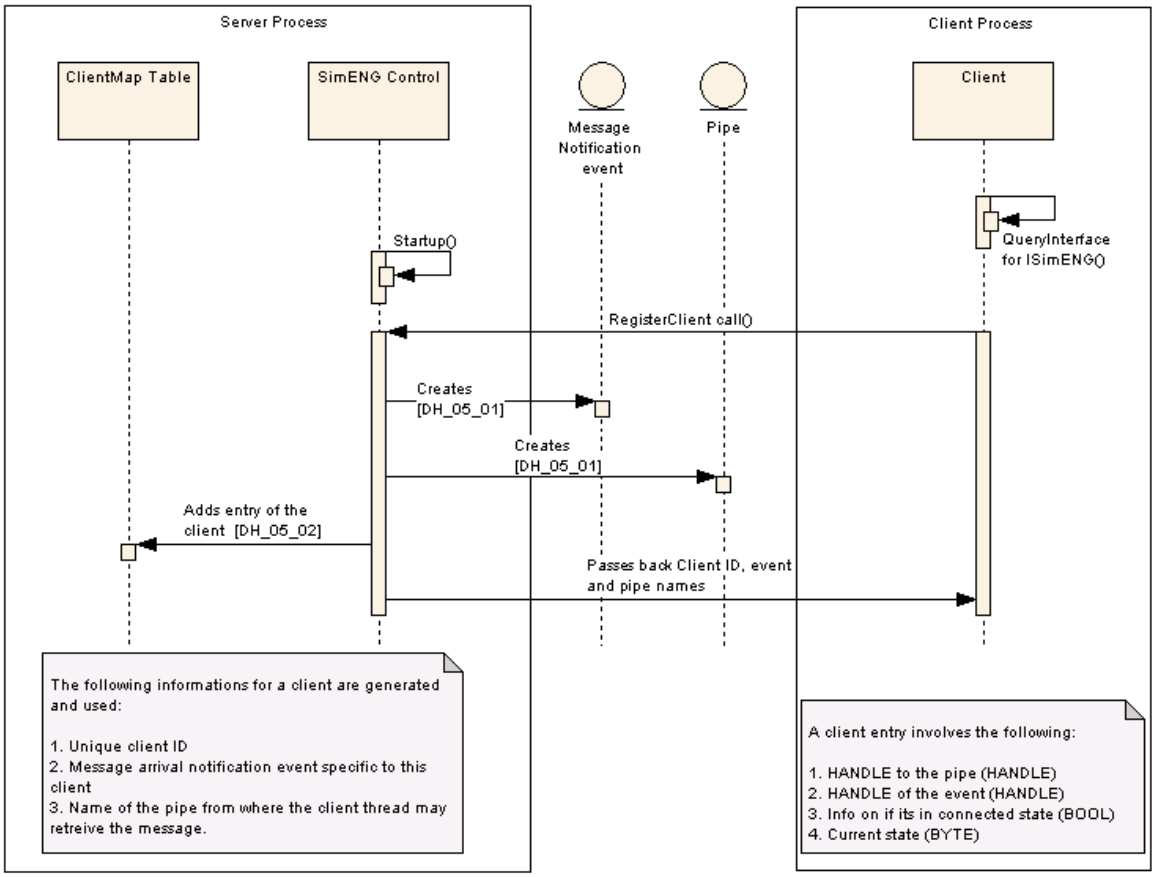


Clearly, it a manifestation of inter process communication with an abstraction on the vehicle bus standard data type. Communication among various instances of the application takes place through an out-of-proc server named as SimENG. The server exposes an API set which a client may use to exploit its functionalities. The server keeps a list of all its clients. When a client calls “SendMessage”, the server adds this message entry into a circular buffer. The message entry saves id of the sender in a parameter of it. SimENG maintains a worker thread which gets activated when the circular buffer is non-empty. It then retrieves the current entry and runs through the client list. In case the present client is the sender, message status is kept as “Tx”, else the default setting is “Rx”. The thread pumps the current message entry into the pipe and signals an event which is an auto-reset one. A thread inside the client process must wait for that. The waiting thread inside the client process then retrieves the message from the pipe.

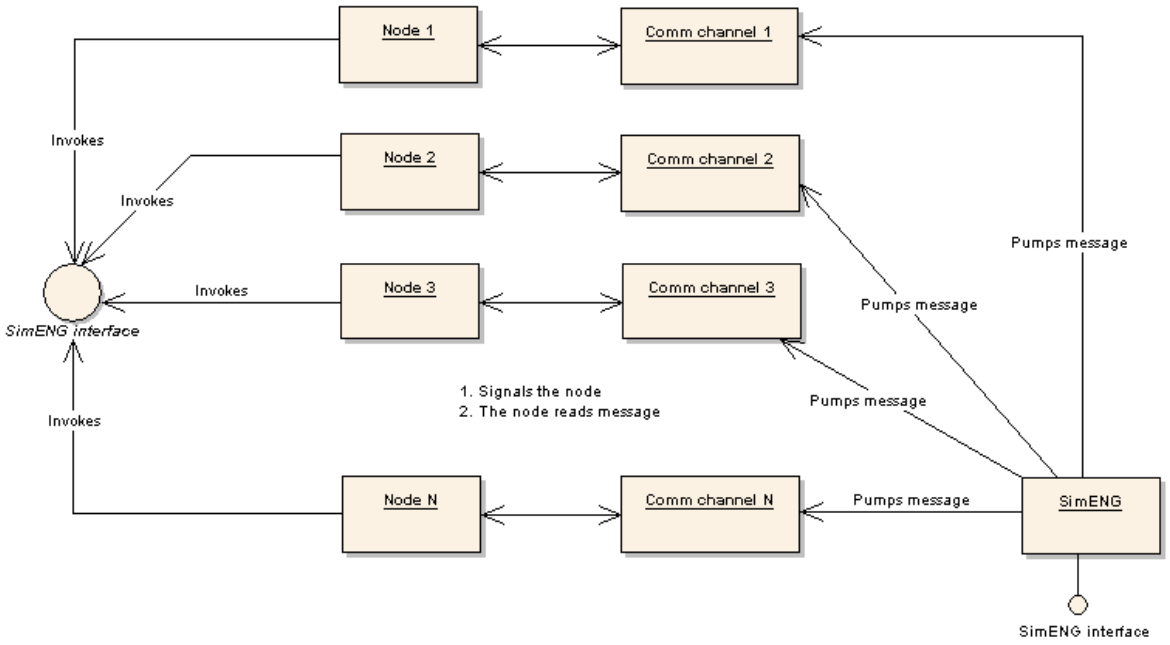
The following diagram shows the registration / unregistration activities of multiple clients in a macroscopic view.

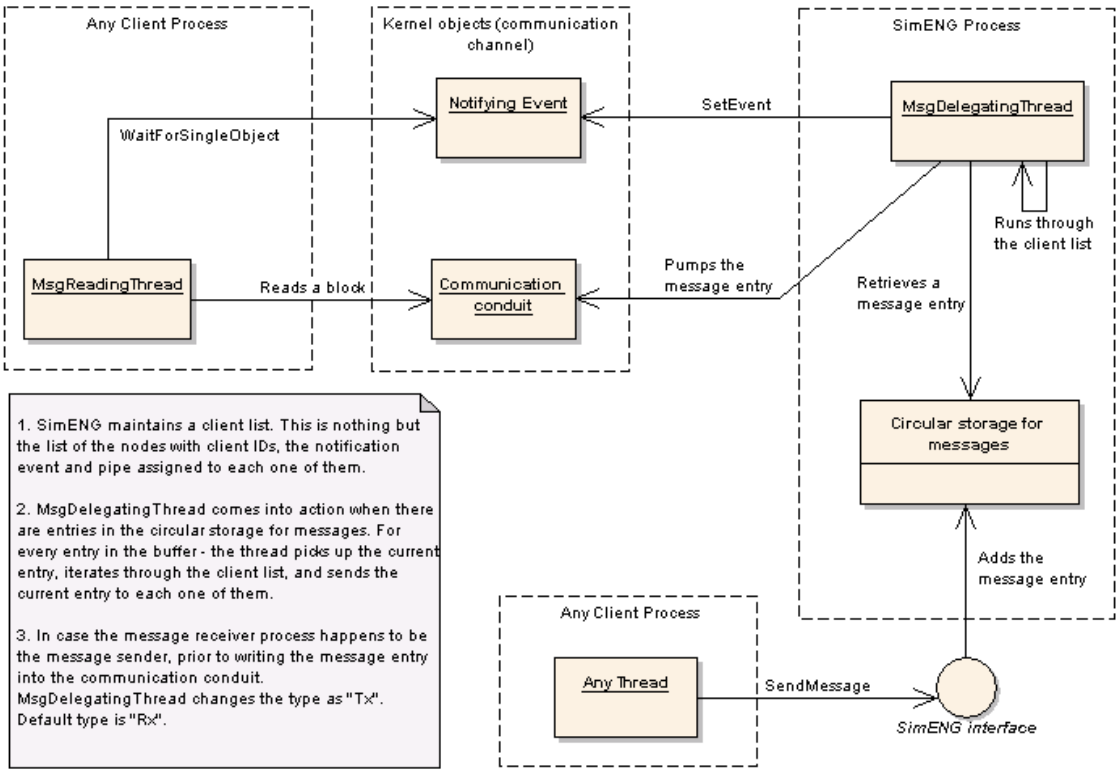


The registration procedure is elaborated in details with the following diagram:



The below diagrams describe the process of frame transmission and dispatching:

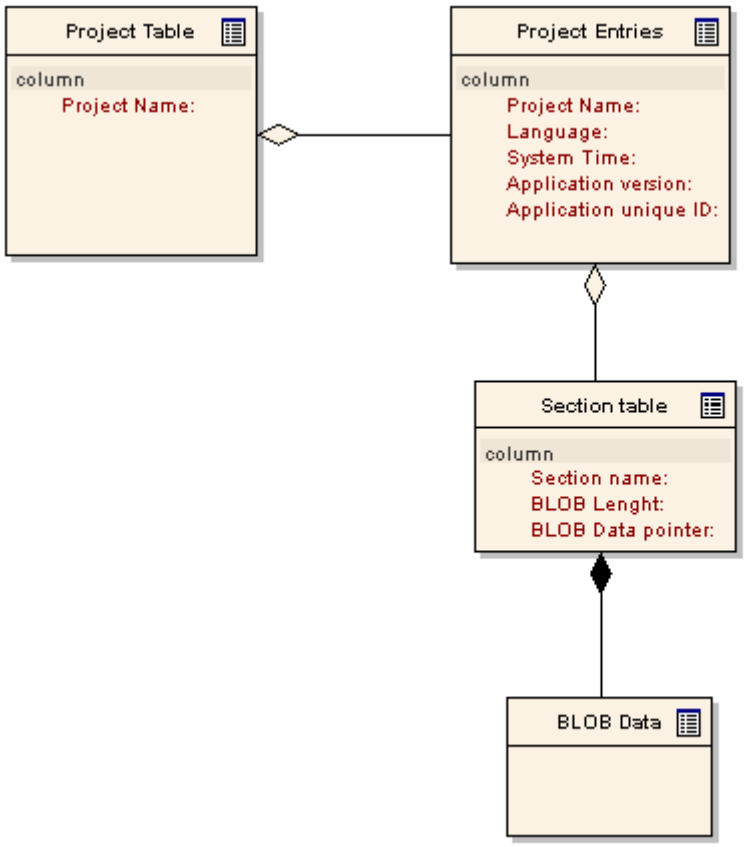




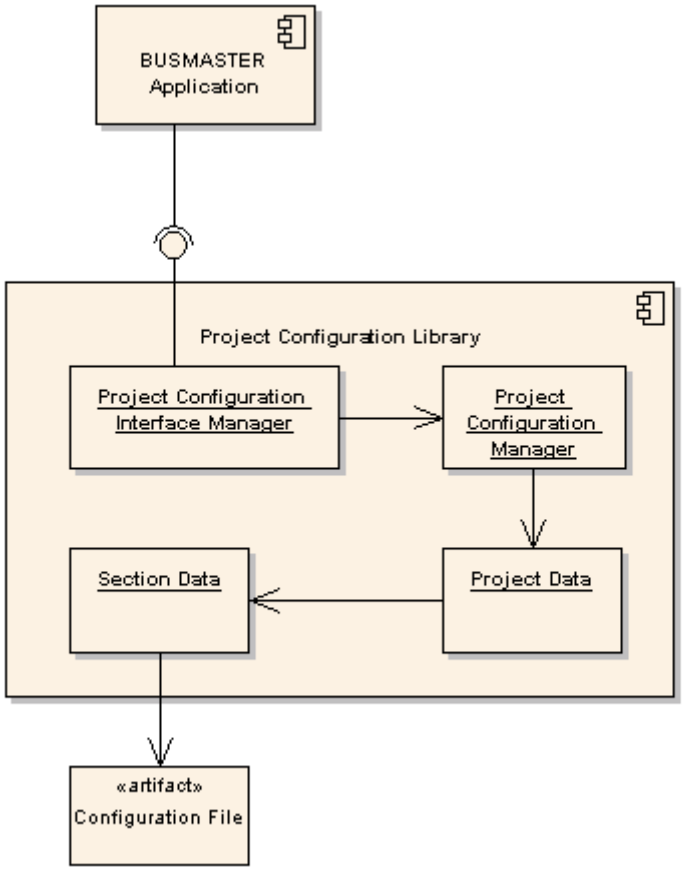
Project Configuration

This module is implemented as a DLL which exports some functions for one or multiple project configuration data retention in a file. The file is called the project configuration file. It's a binary file.

Project data are logically organised to follow a schema alike to database. This means the entry point is a master table which is called 'Project Table' and lists the different project entries. So the primary key of each entry is the project name. This means it is possible to work with configuration data of multiple projects. Each project entry is parameterised with information like language code (ISO 639-3), time and date of last editing, application version and the unique identifier of the application. Configuration data section of a project is maintained in an entity called section table. Each project has its own section table which is identified by the name of the project. A section table is the summation of section entries, which is defined by a section name, the section data pointer and the data length in bytes. Hence, a section is a quasi-unit of a configuration data. This view is depicted in the following data model diagram:

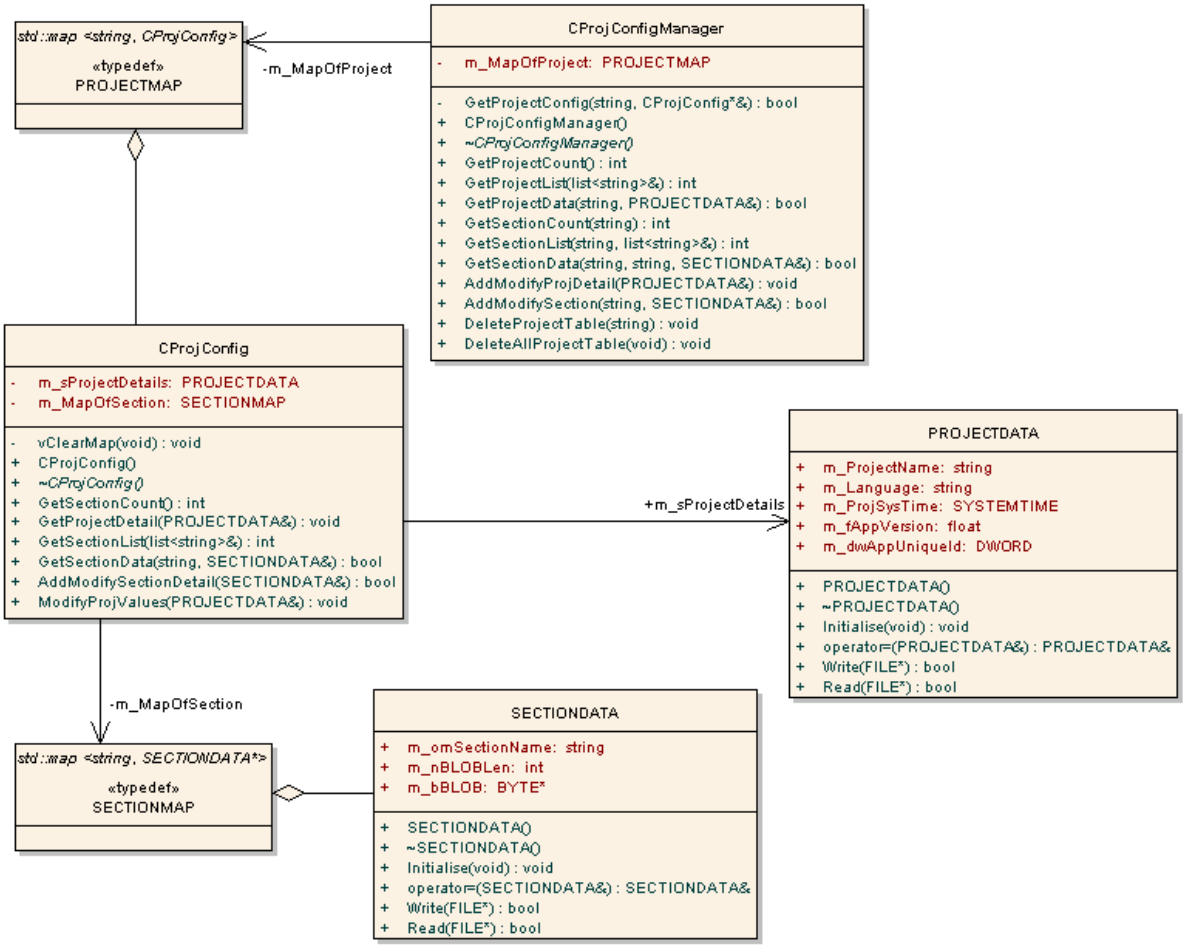


The implementing components are carved based on the aforementioned data model and the interfacing aspect of the library with its client applications or components. A simple API set of exported functions is sufficient for the interfacing requirement. Also, there must be avenues to use both file and database system (should the need arise to realise this in future) for the archiving purpose. Hence, results the first object named as ‘Project Configuration Interface Manager’. Based of the current choice (file or database system), the path forks.



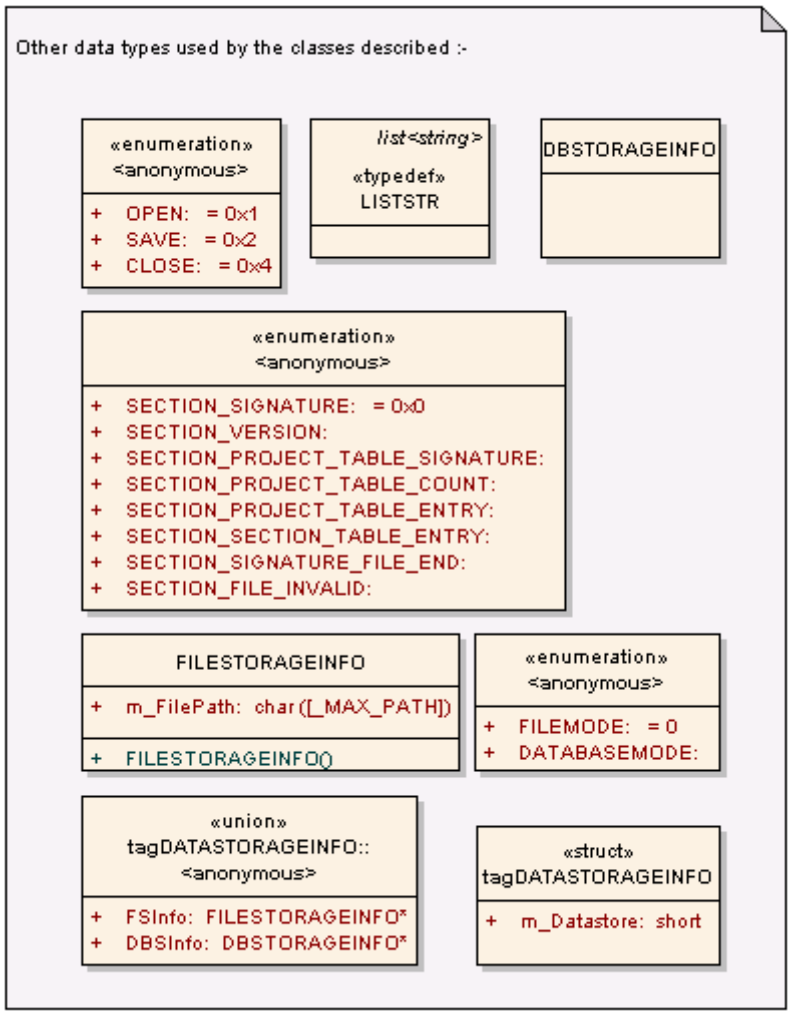
The object ‘Project Configuration Manager’ deals with the project list whereas the entity ‘Project Data’ manages individual project data (the section list) and makes it possible to process section data through ‘Section Data’ object. The four entities are the cornerstones of the Project Configuration library. The component diagram shows this arrangement with an outlining on the data and control flow.

The concrete implementation is described with the following class diagram. Again, the class designing is directed by the data model schema and closely reflect the organizational



aspect.

The other supporting data types are presented

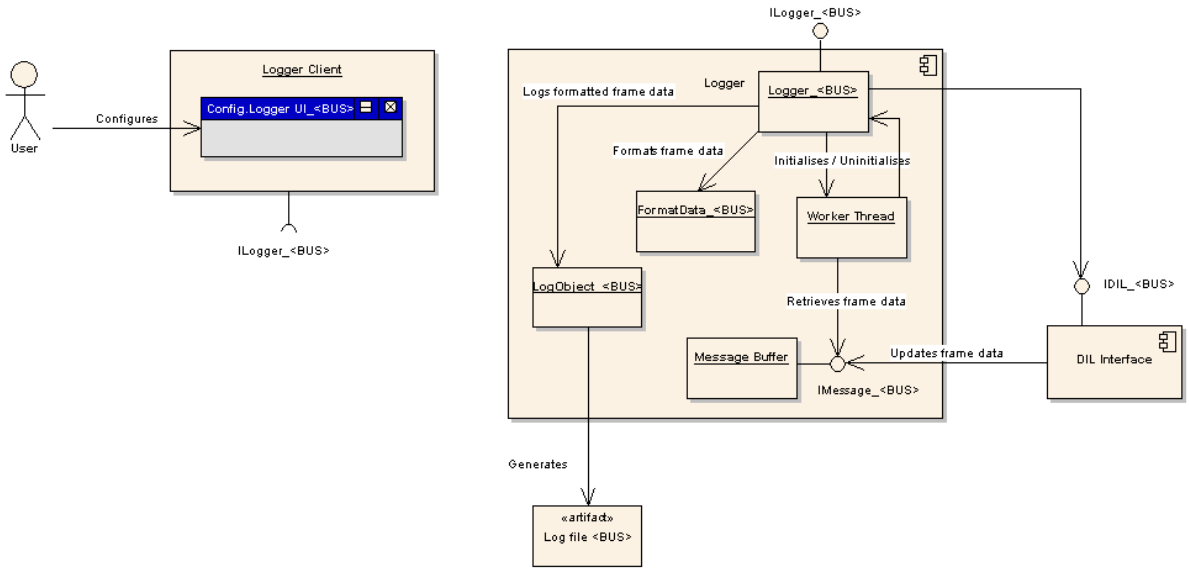


below.

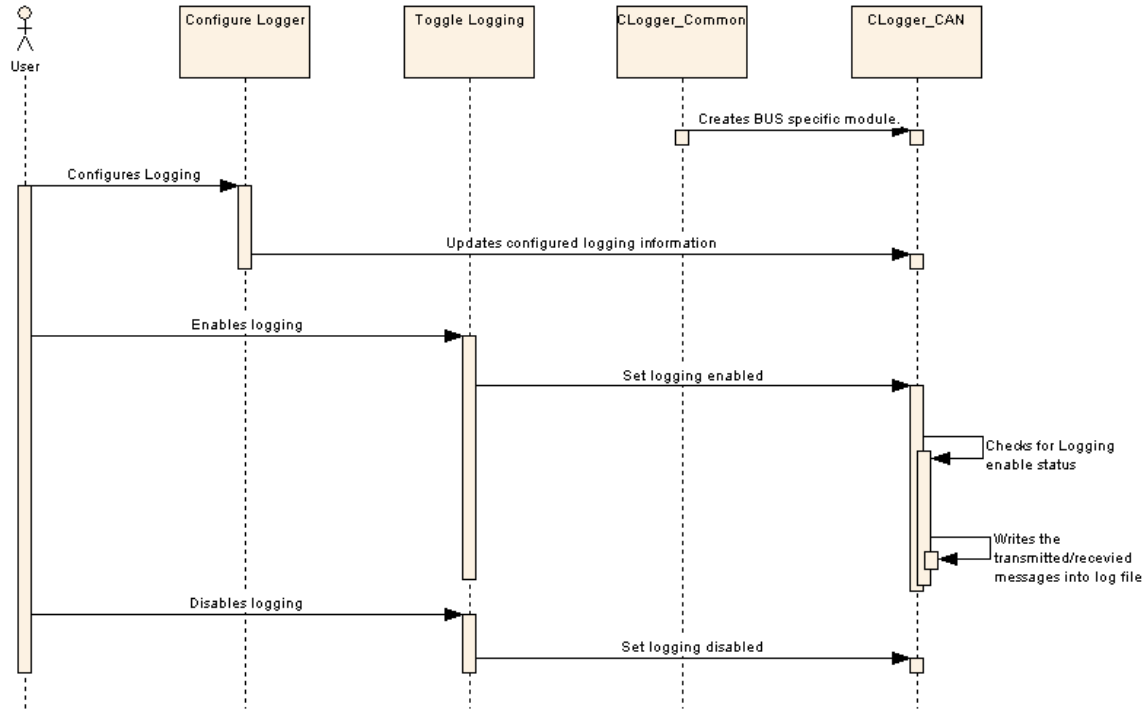
Therefore, design of project configuration library again follows a hybrid design methodology, albeit well rhythmized.

Logger

Data channel abstraction procedure continues to be followed as expected. Logger module receives the client ID from its client. A dedicated worker thread is the engine of the whole mechanism and drives the logging process. It waits for data frame event signals of the buffer; retrieves frame data, formats the same and finally logs the formatted frame data into a log file, through other specific objects acting as agencies. Logging configuration aspect is addressed by an object denoted by Logger_<BUS> (e.g., for CAN the class name is CFrameProcessor_CAN). The following component diagram explains this mechanism. No particular bus name is mentioned, which means the procedure manifests for any vehicle standard support.



The following sequence diagram shows the configuration and logging activation / deactivation process.

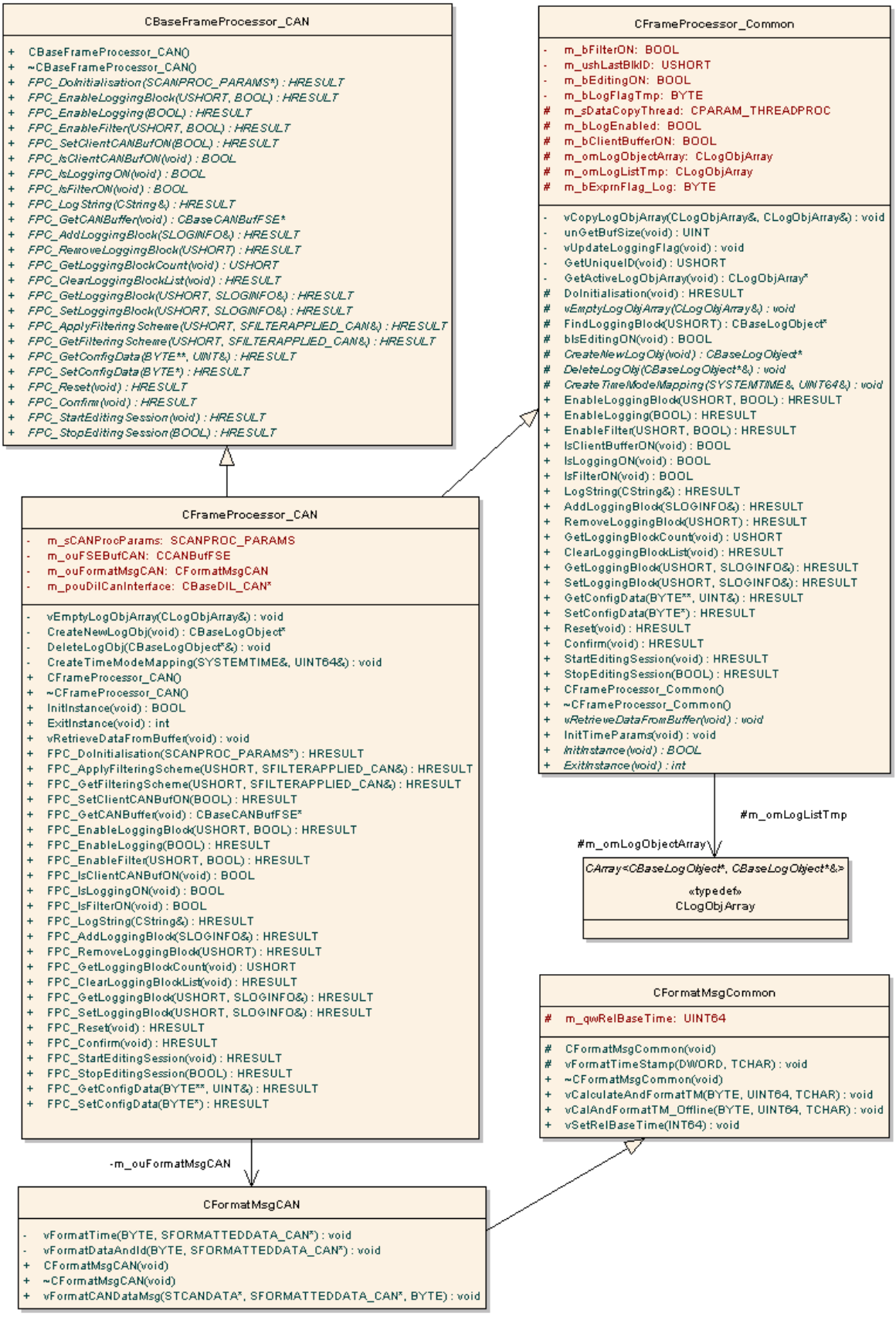


The below class diagram shows the implementation aspect. This largely involves identification of the common activities across all target vehicle bus standards and group them together in terms of classes. Here the emphasis is on reuse through inheritance and encapsulation through proper abstraction. The entity ‘Logger_<BUS>’, in all its possible manifestations (e.g., Logger_CAN, Logger_J1939, Logger_FlexRay etc) exhibits the same characterisation of activity which is – waiting for a data event from the message buffer, retrieving frame data, formatting the same according to the present formatting conditions imposed and logging the resultant string into the target file or file list. The services rendered can easily fall under a distinct category of their own. Unique operations for a particular vehicle bus are the frame data being dealt with, the formatting intricacy and a few related subtle operations.

The inheritance is governed by the following directives derived from the above analysis:

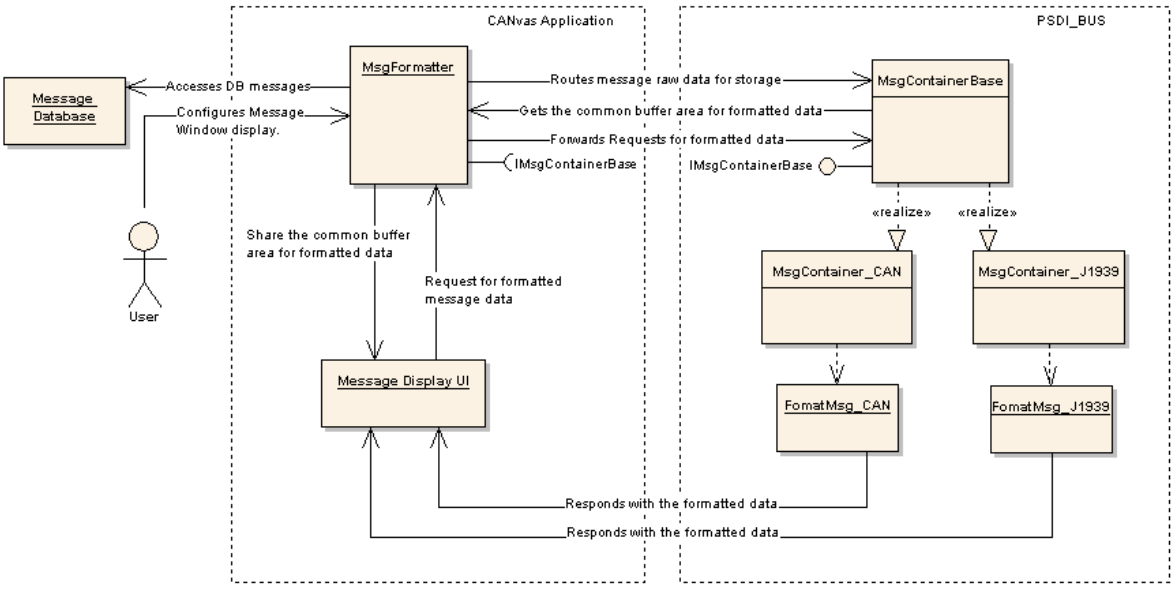
- Logger_<BUS> class is inherited from the interface virtual class

- The common or generic activities translated into another class (to be called CFrameProcessor_Common)
- CFrameProcessor_Common drives the whole mechanism and Logger_<BUS> is inherited from it too.
- The unique operations are prototyped as generic methods and hence can be invoked from CFrameProcessor_Common. They are declared pure virtual and hence the derived class must implement the same.

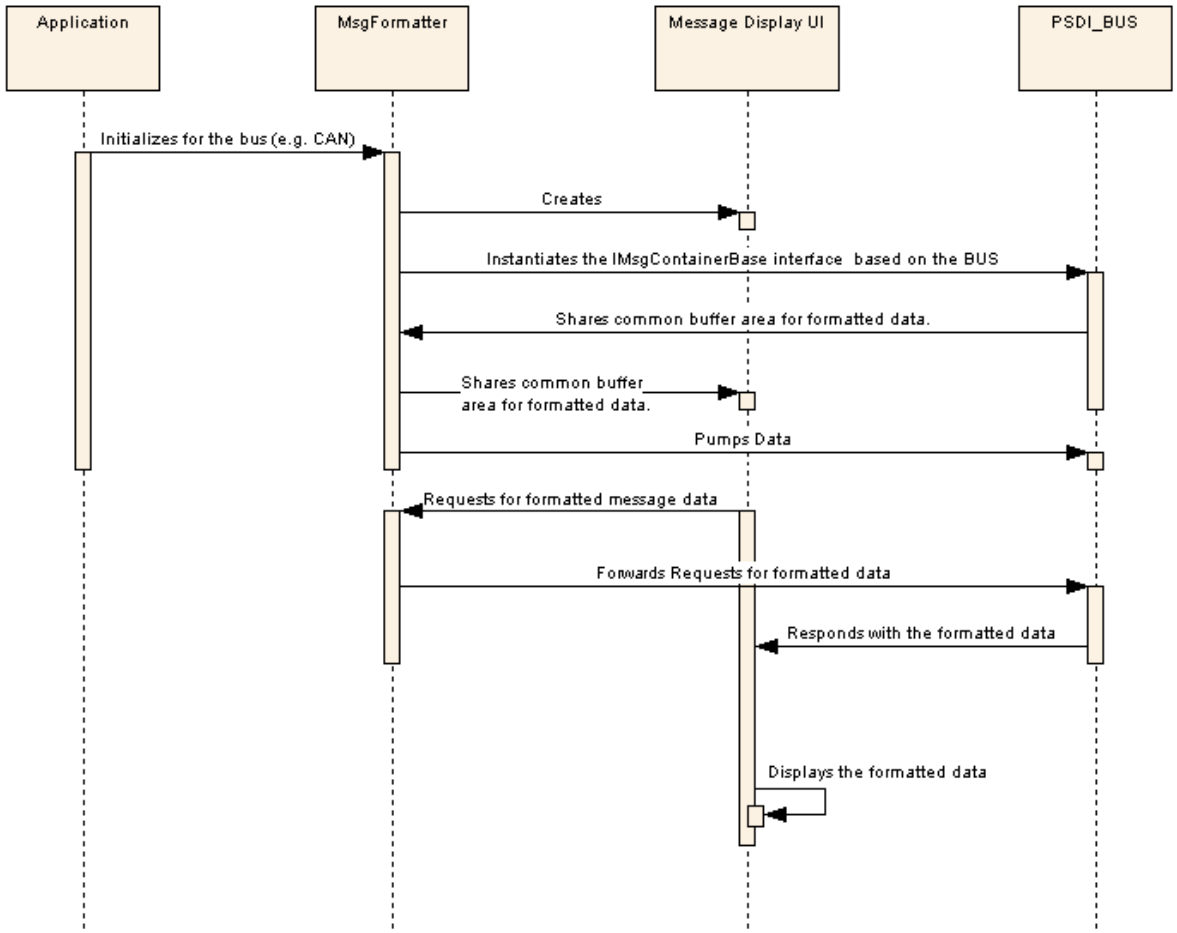

CBaseFrameProcessor_CAN
+ CBaseFrameProcessor_CAN()
+ ~CBaseFrameProcessor_CAN()
+ FPC_DoInitialisation(SCANPROC_PARAMS*) : HRESULT
+ FPC_EnableLoggingBlock(USHORT, BOOL) : HRESULT
+ FPC_EnableLogging(BOOL) : HRESULT
+ FPC_EnableFilter(USHORT, BOOL) : HRESULT
+ FPC_SetClientCANBufON(BOOL) : HRESULT
+ FPC_IsClientCANBufON(void) : BOOL
+ FPC_IsLoggingON(void) : BOOL
+ FPC_IsFilterON(void) : BOOL
+ FPC_LogString(CString&) : HRESULT
+ FPC_GetCANBuffer(void) : CBaseCANBufFSE*
+ FPC_AddLoggingBlock(SLOGINFO&) : HRESULT
+ FPC_RemoveLoggingBlock(USHORT) : HRESULT
+ FPC_GetLoggingBlockCount(void) : USHORT
+ FPC_ClearLoggingBlockList(void) : HRESULT
+ FPC_GetLoggingBlock(USHORT, SLOGINFO&) : HRESULT
+ FPC_SetLoggingBlock(USHORT, SLOGINFO&) : HRESULT
+ FPC_ApplyFilteringScheme(USHORT, SFILTERAPPLIED_CAN&) : HRESULT
+ FPC_GetFilteringScheme(USHORT, SFILTERAPPLIED_CAN&) : HRESULT
+ FPC_GetConfigData(BYTE**, UINT&) : HRESULT
+ FPC_SetConfigData(BYTE*) : HRESULT
+ FPC_Reset(void) : HRESULT
+ FPC_Confirm(void) : HRESULT
+ FPC_StartEditing Session(void) : HRESULT
+ FPC_StopEditing Session(BOOL) : HRESULT
CFrameProcessor_Common
- m_bFilterON: BOOL
- m_ushLastBlkID: USHORT
- m_bEditingON: BOOL
- m_bLogFlagTmp: BYTE
m_sDataCopyThread: CPARAM_THREADPROC
m_bLogEnabled: BOOL
m_bClientBufferON: BOOL
m_omLogObjectArray: CLogObjArray
m_omLogListTmp: CLogObjArray
m_bExprnFlag_Log: BYTE
- vCopyLogObjArray(CLogObjArray&, CLogObjArray&) : void
- unGetBufSize(void) : UINT
- vUpdateLoggingFlag(void) : void
- GetUniqueID(void) : USHORT
- GetActiveLogObjArray(void) : CLogObjArray*
DoInitialisation(void) : HRESULT
vEmptyLogObjArray(CLogObjArray&) : void
FindLoggingBlock(USHORT) : CBaseLogObject*
bIsEditingON(void) : BOOL
CreateNewLogObj(void) : CBaseLogObject*
DeleteLogObj(CBaseLogObject*&) : void
CreateTimeModeMapping(SYSTEMTIME&, UINT64&) : void
+ EnableLoggingBlock(USHORT, BOOL) : HRESULT
+ EnableLogging(BOOL) : HRESULT
+ EnableFilter(USHORT, BOOL) : HRESULT
+ IsClientBufferON(void) : BOOL
+ IsLoggingON(void) : BOOL
+ IsFilterON(void) : BOOL
+ LogString(CString&) : HRESULT
+ AddLoggingBlock(SLOGINFO&) : HRESULT
+ RemoveLoggingBlock(USHORT) : HRESULT
+ GetLoggingBlockCount(void) : USHORT
+ ClearLoggingBlockList(void) : HRESULT
+ GetLoggingBlock(USHORT, SLOGINFO&) : HRESULT
+ SetLoggingBlock(USHORT, SLOGINFO&) : HRESULT
+ GetConfigData(BYTE**, UINT&) : HRESULT
+ SetConfigData(BYTE*) : HRESULT
+ Reset(void) : HRESULT
+ Confirm(void) : HRESULT
+ StartEditingSession(void) : HRESULT
+ StopEditingSession(BOOL) : HRESULT
+ CFrameProcessor_Common()
+ ~CFrameProcessor_Common()
+ vRetrieveDataFromBuffer(void) : void
+ InitTimeParams(void) : void
+ InitInstance(void) : BOOL
+ ExitInstance(void) : int
CFrameProcessor_CAN
- m_sCANProcParams: SCANPROC_PARAMS
- m_ouFSEBufCAN: CCANBufFSE
- m_ouFormatMsgCAN: CFormatMsgCAN
- m_pouDilCanInterface: CBaseDIL_CAN*
- vEmptyLogObjArray(CLogObjArray&) : void
- CreateNewLogObj(void) : CBaseLogObject*
- DeleteLogObj(CBaseLogObject*&) : void
- CreateTimeModeMapping(SYSTEMTIME&, UINT64&) : void
+ CFrameProcessor_CAN()
+ ~CFrameProcessor_CAN()
+ InitInstance(void) : BOOL
+ ExitInstance(void) : int
+ vRetrieveDataFromBuffer(void) : void
+ FPC_DoInitialisation(SCANPROC_PARAMS*) : HRESULT
+ FPC_ApplyFilteringScheme(USHORT, SFILTERAPPLIED_CAN&) : HRESULT
+ FPC_GetFilteringScheme(USHORT, SFILTERAPPLIED_CAN&) : HRESULT
+ FPC_SetClientCANBufON(BOOL) : HRESULT
+ FPC_GetCANBuffer(void) : CBaseCANBufFSE*
+ FPC_EnableLoggingBlock(USHORT, BOOL) : HRESULT
+ FPC_EnableLogging(BOOL) : HRESULT
+ FPC_EnableFilter(USHORT, BOOL) : HRESULT
+ FPC_IsClientCANBufON(void) : BOOL
+ FPC_IsLoggingON(void) : BOOL
+ FPC_IsFilterON(void) : BOOL
+ FPC_LogString(CString&) : HRESULT
+ FPC_AddLoggingBlock(SLOGINFO&) : HRESULT
+ FPC_RemoveLoggingBlock(USHORT) : HRESULT
+ FPC_GetLoggingBlockCount(void) : USHORT
+ FPC_ClearLoggingBlockList(void) : HRESULT
+ FPC_GetLoggingBlock(USHORT, SLOGINFO&) : HRESULT
+ FPC_SetLoggingBlock(USHORT, SLOGINFO&) : HRESULT
+ FPC_Reset(void) : HRESULT
+ FPC_Confirm(void) : HRESULT
+ FPC_StartEditingSession(void) : HRESULT
+ FPC_StopEditingSession(BOOL) : HRESULT
+ FPC_GetConfigData(BYTE**, UINT&) : HRESULT
+ FPC_SetConfigData(BYTE*) : HRESULT
#m_omLogListTmp
#m_omLogObjectArray
CArray<CBaseLogObject*, CBaseLogObject*&>
«typedef»
CLogObjArray
CFormatMsgCommon
m_qwRelBaseTime: UINT64
CFormatMsgCommon(void)
vFormatTimeStamp(DWORD, TCHAR) : void
+ ~CFormatMsgCommon(void)
+ vCalculateAndFormatTM(BYTE, UINT64, TCHAR) : void
+ vCalAndFormatTM_Offline(BYTE, UINT64, TCHAR) : void
+ vSetRelBaseTime(INT64) : void
-m_ouFormatMsgCAN
CFormatMsgCAN
- vFormatTime(BYTE, SFORMATTEDDATA_CAN*) : void
- vFormatDataAndId(BYTE, SFORMATTEDDATA_CAN*) : void
+ CFormatMsgCAN(void)
+ ~CFormatMsgCAN(void)
+ vFormatCANDataMsg(STCANDATA*, SFORMATTEDDATA_CAN*, BYTE) : void

Message Window

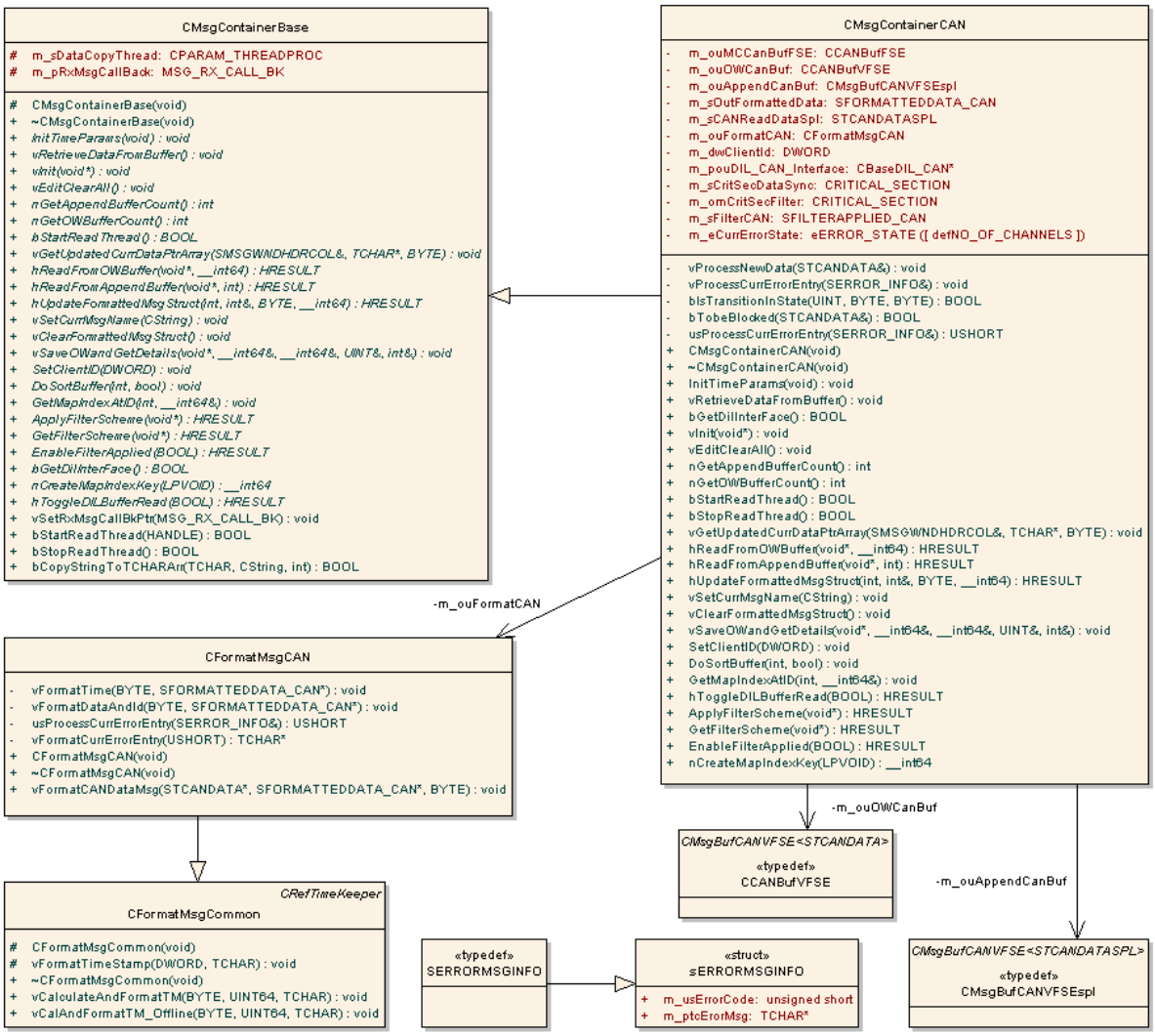
Modus operandi of message window is similar to Bus statistics or logger modules. Here also the general functional part and the vehicle standard specific routines are grouped asunder, into non-cohesive units. From deployment viewpoint the general functional sub-module resides within the application whereas the data formatting routines are realised in a DLL called PSDI_<BUS>. The below component diagram shows the mechanism.



The behavioural aspect of message window module is outlined with the following sequence diagram which shows the initialisation and normal usage instance.



The class design approach is similar to the logger module. The class diagram is presented below:

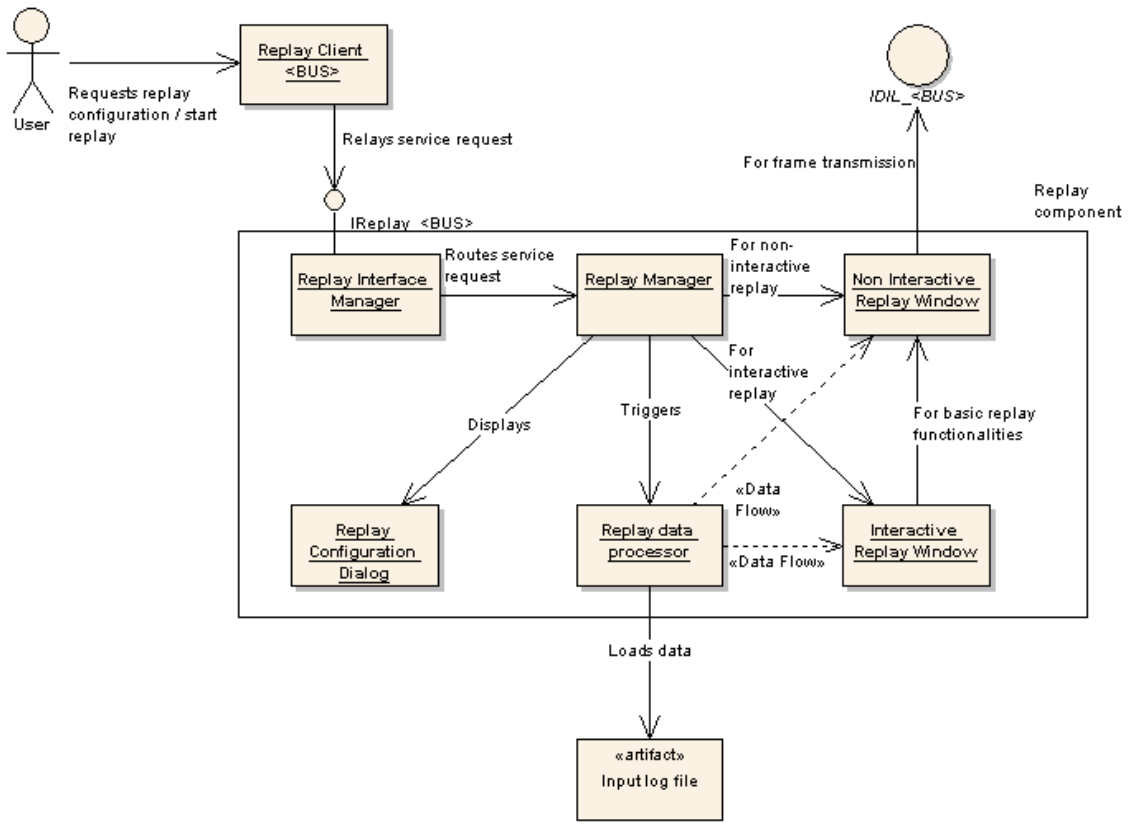


Filter

<TBD>

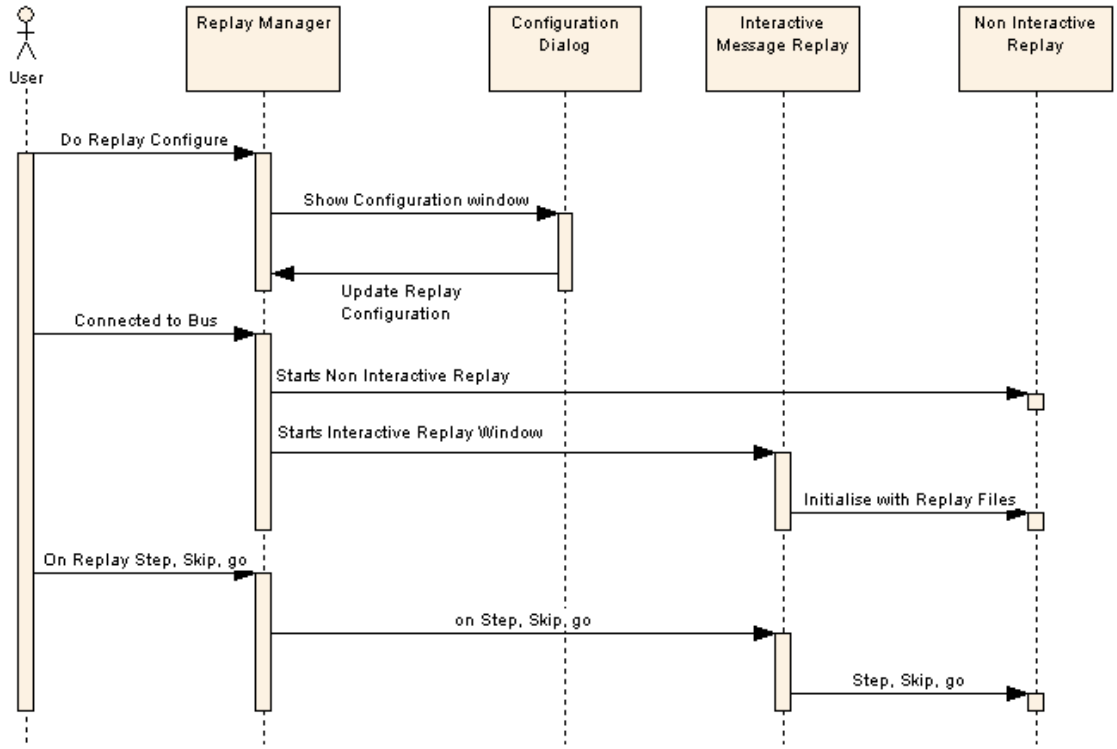
Session Replay

For both the configuration and replay session running the trigger comes from the user. The replay client delegates the service requests to the replay module through its interface. A replay interface manager manages the interface functions and routes the calls appropriately to the replay manager which employs the appropriate sub-modules to carry out the desired tasks. Replay data processor entity accesses the input log file and loads its' data to be treated as the working data set of the replay session. Non-interactive replay window object provides the fundamental replay functionalities such as 'step', 'go', 'skip' and accesses IDIL_<BUS> to exploit the frame transmission functionality of the later.

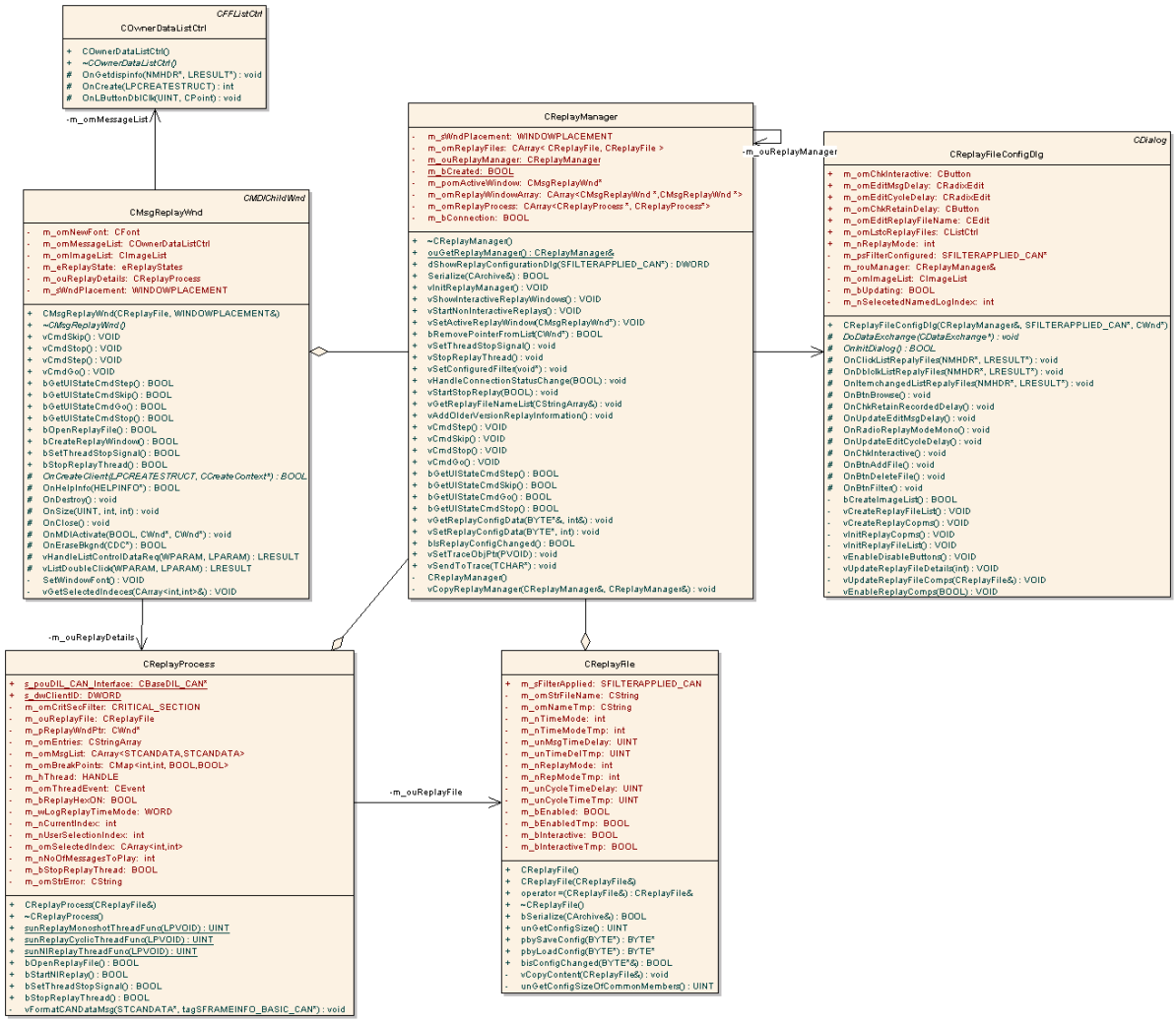


As the feature set of interactive replay window object is a superset of the other replay window, the former accesses the later for its execution. Both the windows are passed the data contained in replay data processor object as the input data from the replay manager.

This is also depicted in the following sequence diagram.

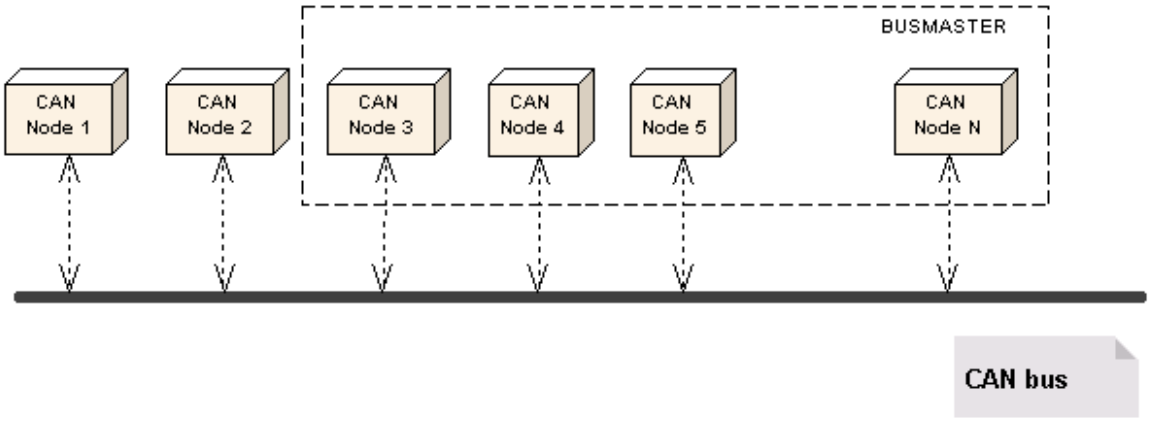


The concrete implementation takes place by a class arrangement described by the following class diagram.

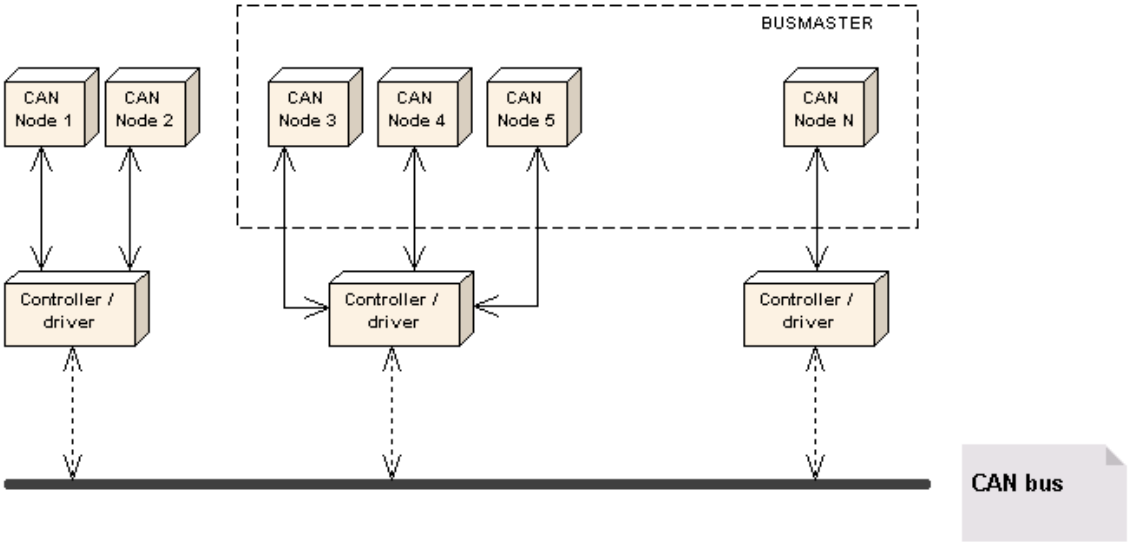


Node Simulation

Design description of node simulation starts with a short prologue on a typical node ensemble and the role of BUSMASTER in the simulation.

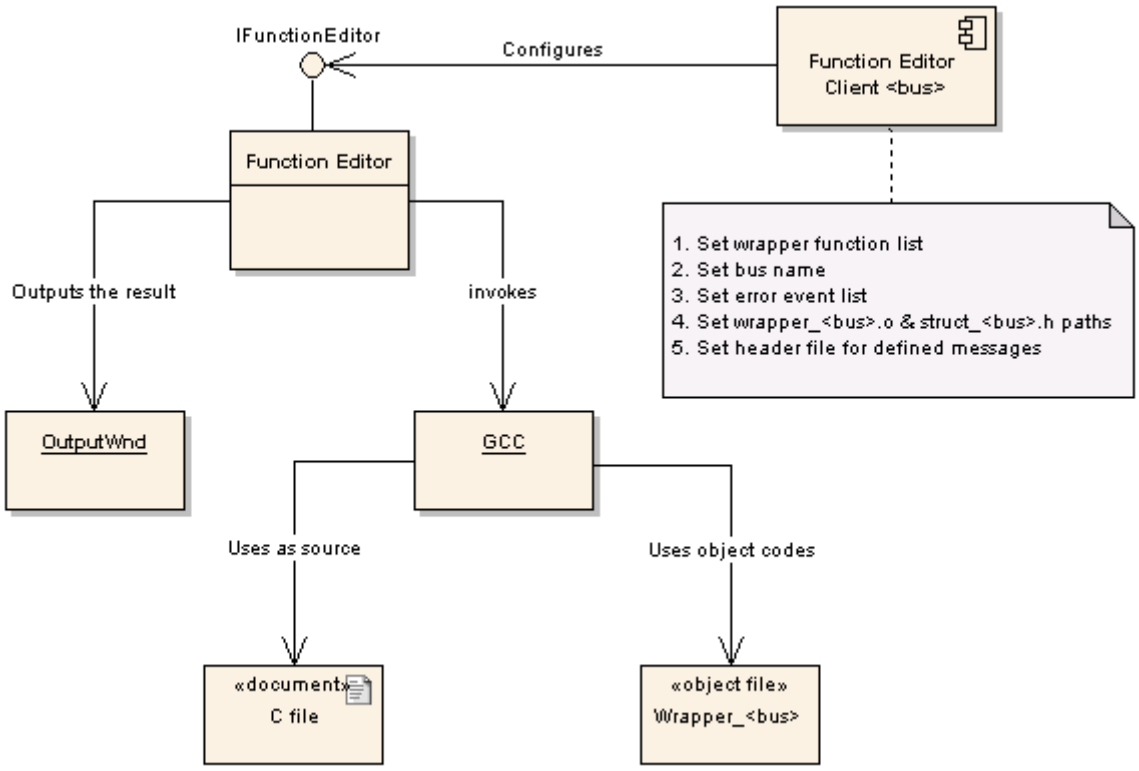


The above diagram exemplifies a cluster of CAN nodes from user's perspective and how BUSMASTER is going to simulate the chosen nodes.



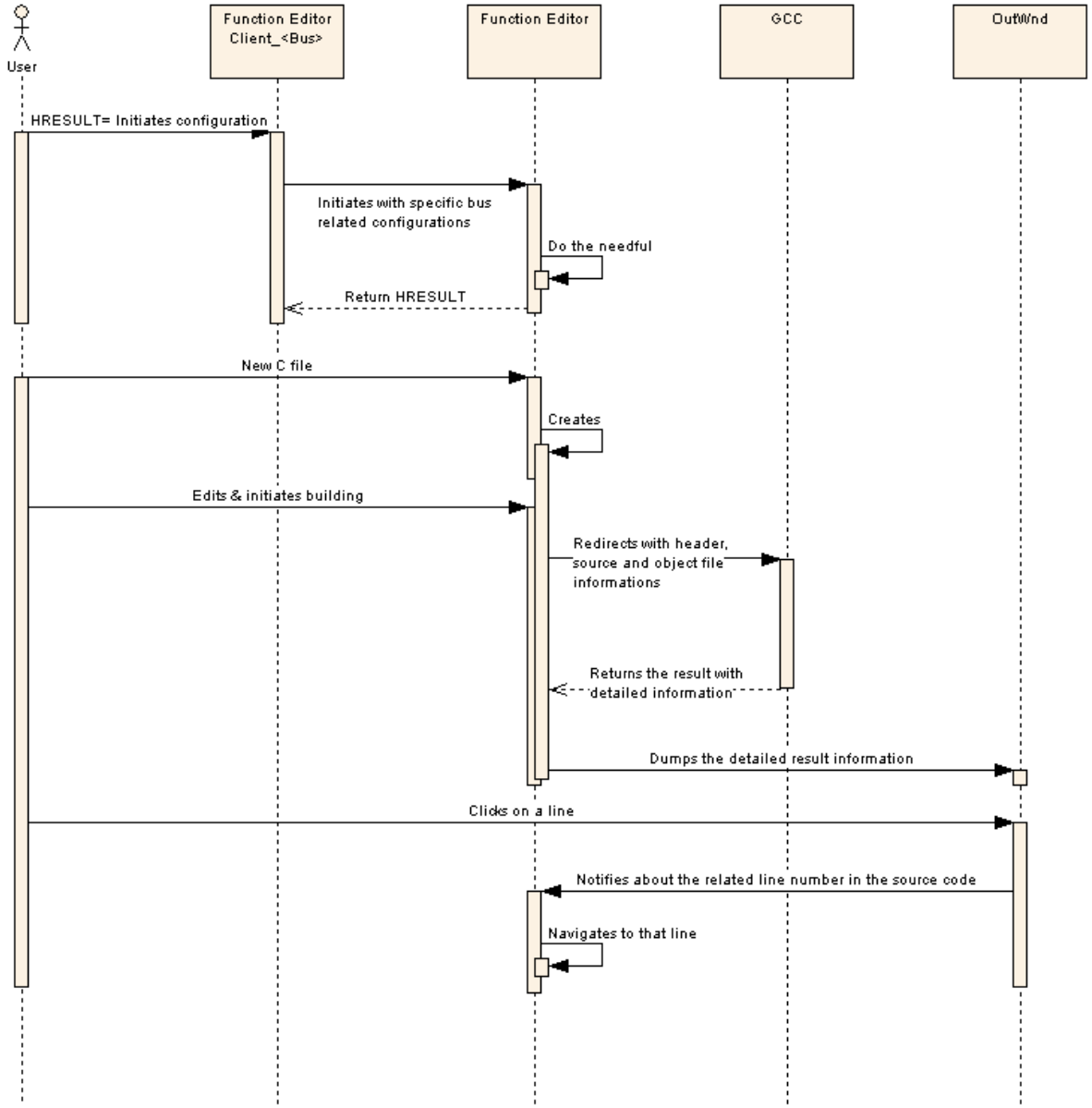
Above is shown what it means from the developer's perspective to simulate one or multiple nodes, from a closer view with the visibility of the bus controllers. Some controllers can simulate multiple nodes. In case it doesn't BUSMASTER does it. Each node simulation is realised by a DLL running in the context of the application. Also, every node should be able to carry out certain bus specific operations (like frame transmission, connecting / disconnecting from the network etc) and hence a suitable API with required services must be available for the DLL. The services must best be rendered by the application.

Function editor module makes it possible to generate the DLL from a user-edited 'C' file. The below diagram shows the building process:

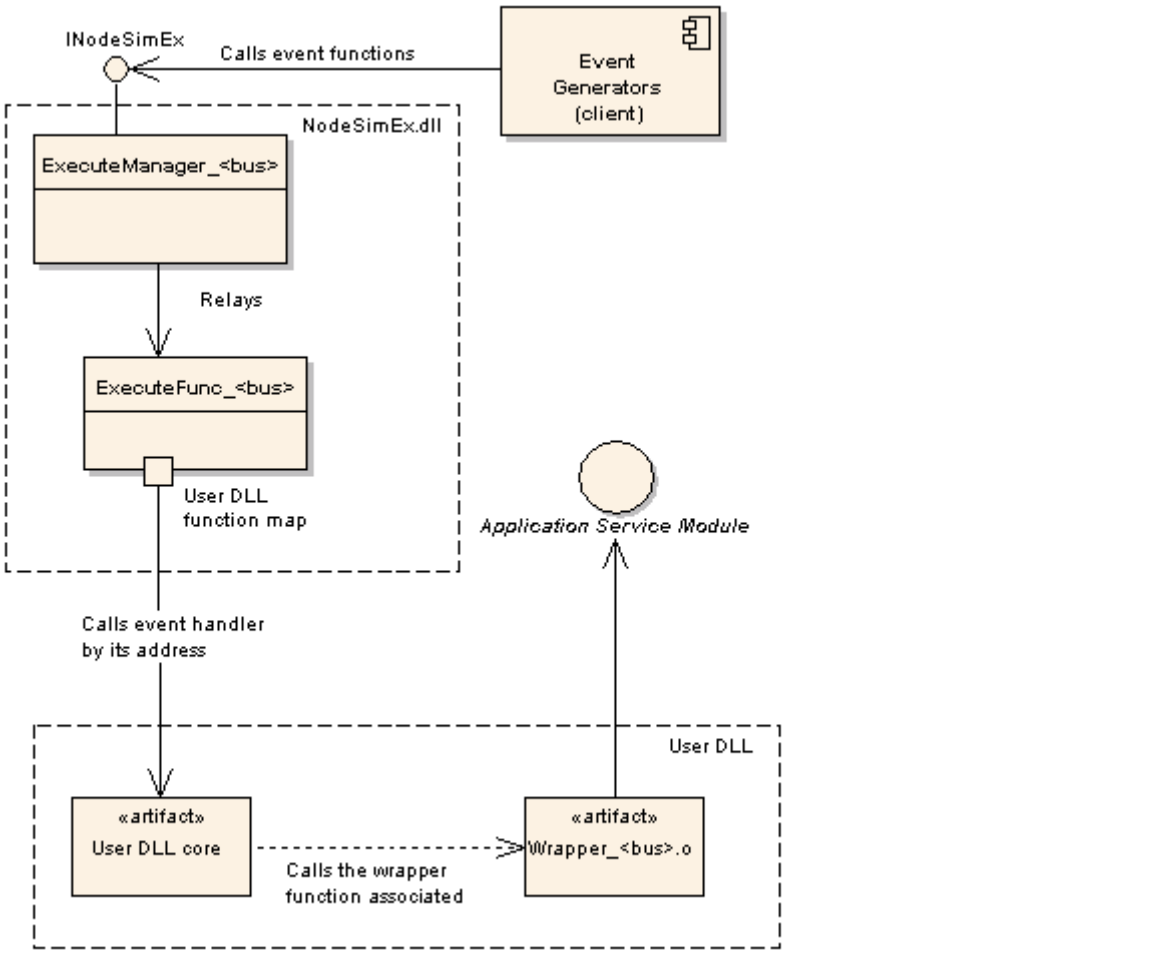


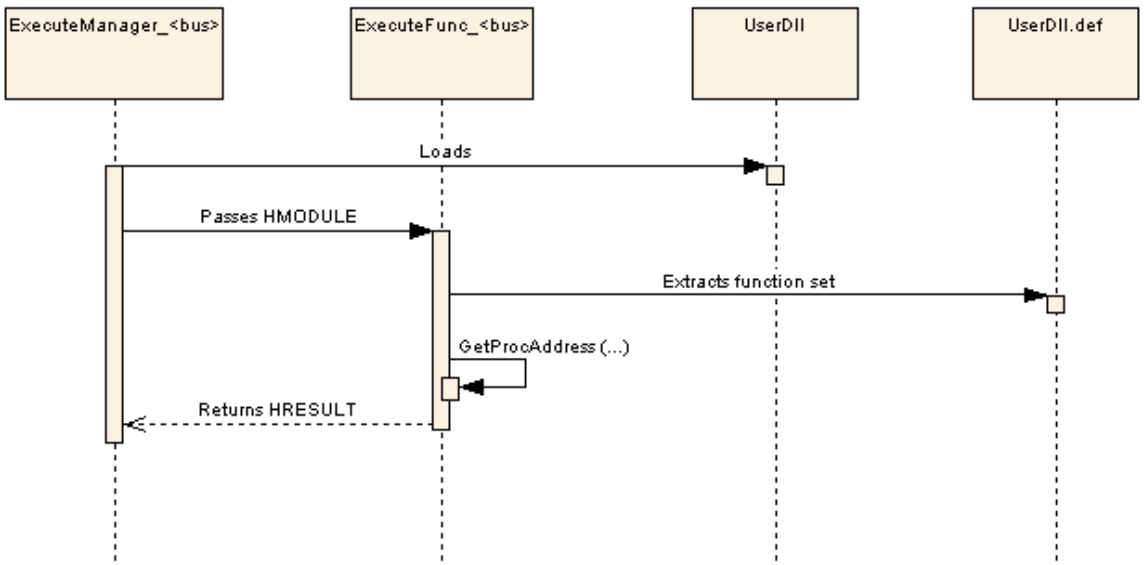
Please note that an object file is necessary for extracting the application's API set for a particular bus.

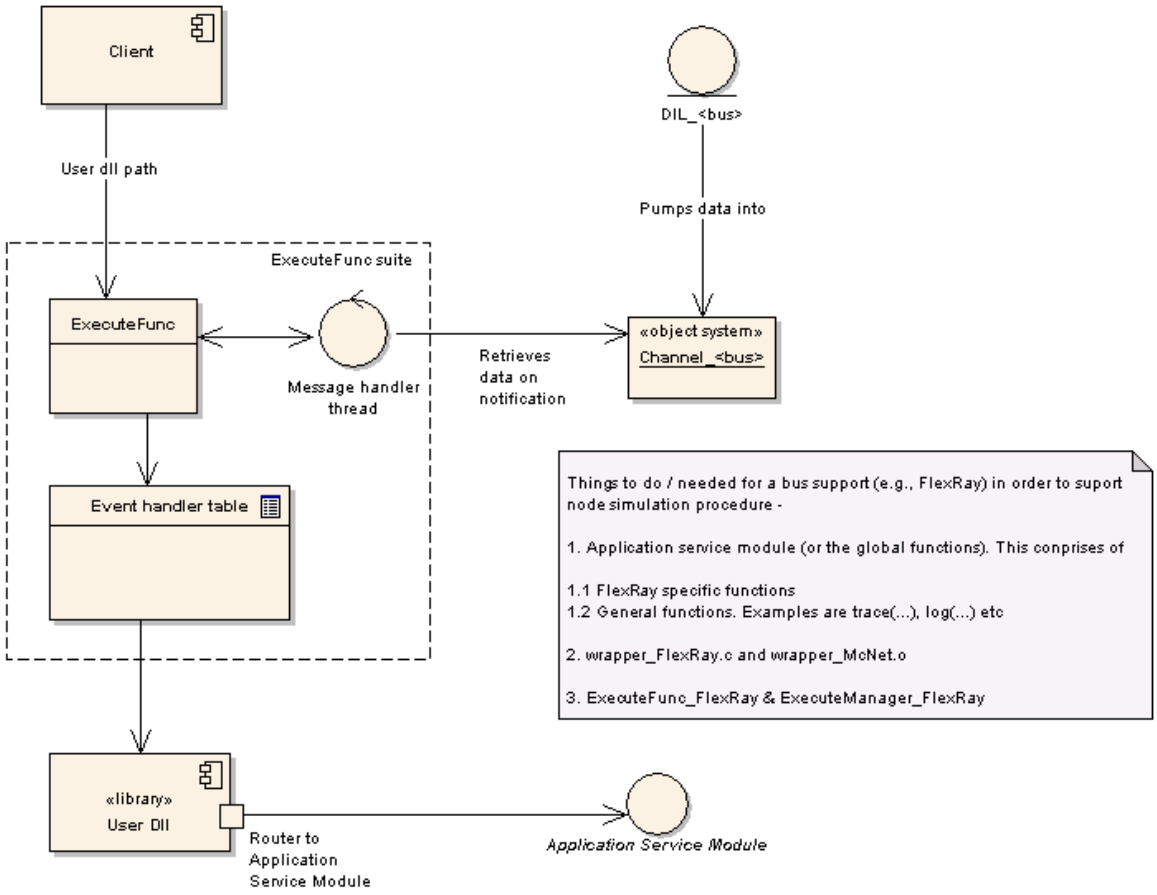
For Function Editor the primary bus specific entities are application API set and database information. So it is possible to implement the function editor in a generic way and dynamically configure it by the respective bus specific client. The below diagram describes this process.



The execution model of node simulation is depicted below. It is the application that mobilises the activity with the reception of the handlers (message, error event etc).


INodeSimEx
Calls event functions
Event Generators (client)
NodeSimEx.dll
ExecuteManager_<bus>
Relays
ExecuteFunc_<bus>
User DLL function map
Application Service Module
Calls event handler by its address
User DLL
«artifact» User DLL core
Calls the wrapper function associated
«artifact» Wrapper_<bus>.o


ExecuteManager_<bus>
ExecuteFunc_<bus>
UserDll
UserDll.def
Loads
Passes HMODULE
Extracts function set
GetProcAddress (...)
Returns HRESULT



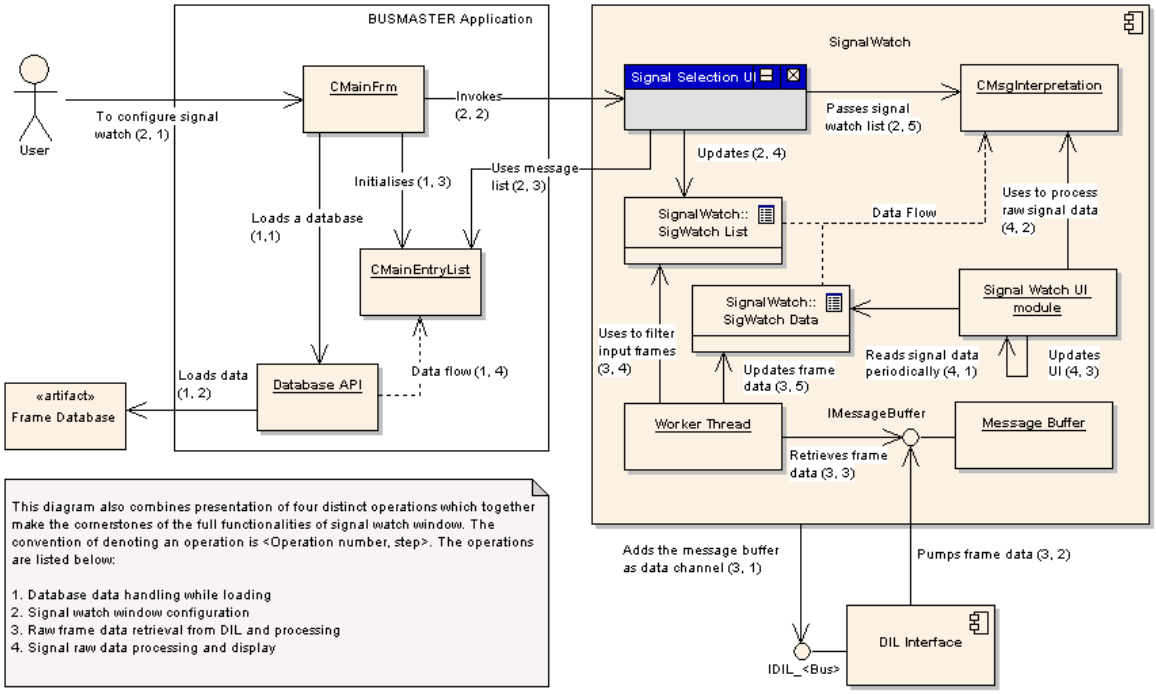
Frame Transmission

<TBD>

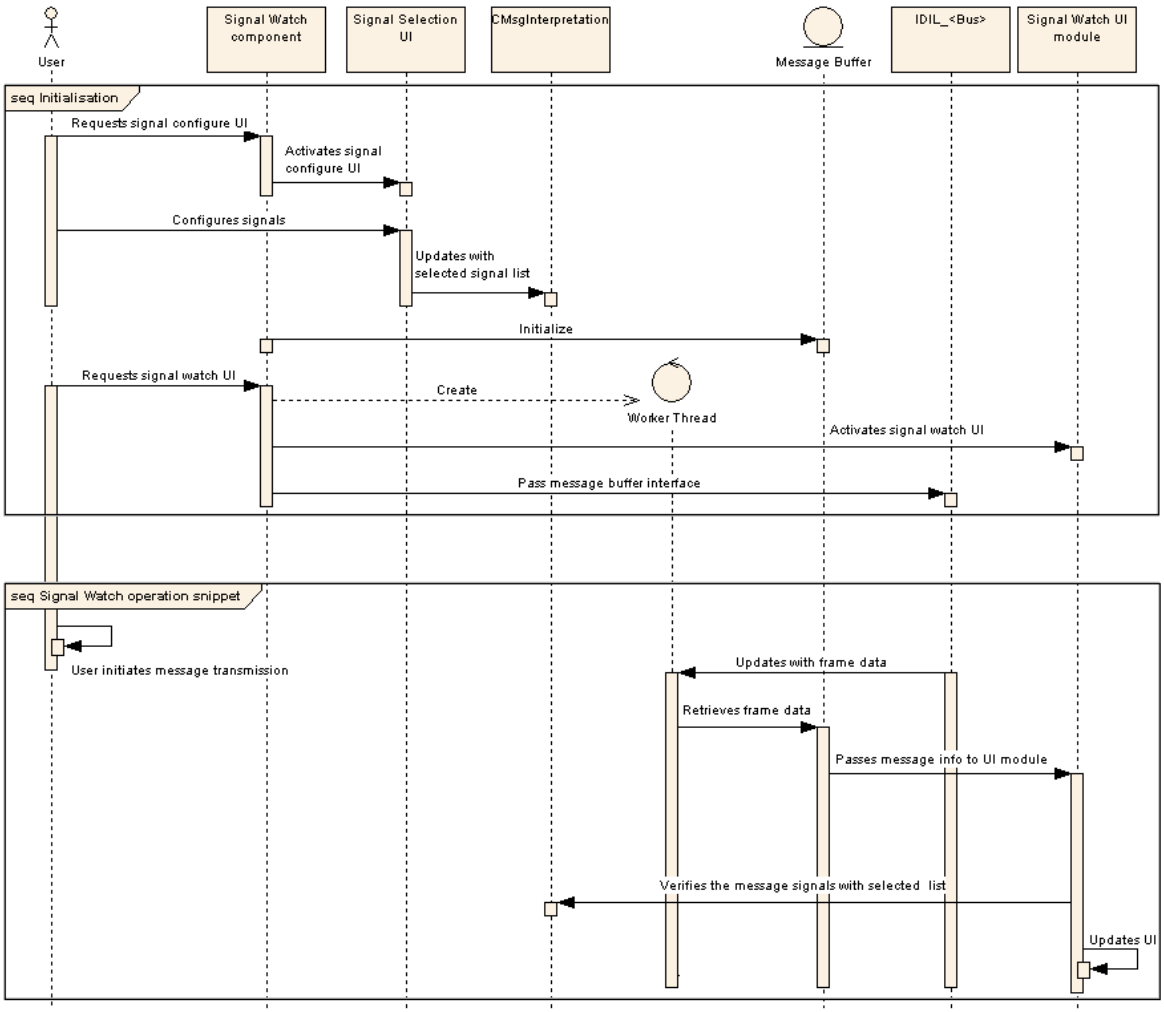
Signal Watch Window

The modus operandi of this component, like other modules viz. bus statistics, message window etc, centres around the already defined concept of abstract data channelling. The only addendum is usage of the loaded message database which is part and parcel of the usage of signal watch window. In fact, without any associated message database, usage of this module bears no significance. So, configuration is an absolute prerequisite.

Signal Watch Window operations covering both the configuration and signal watching activities are explained with the following component diagram with necessary annotations.

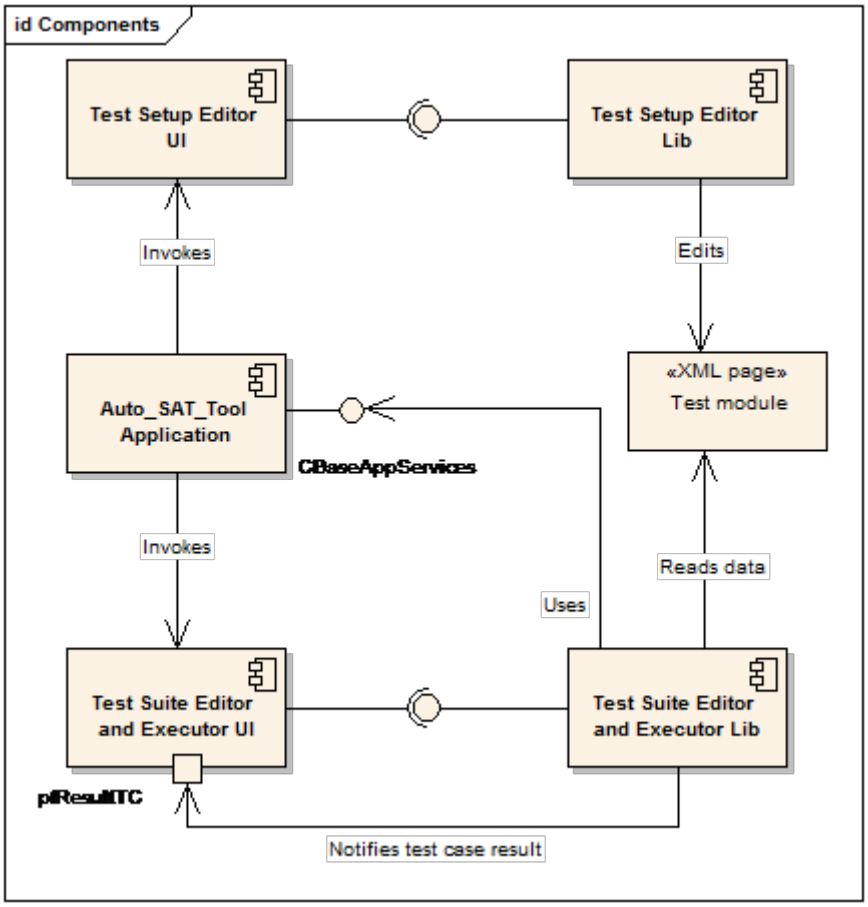


The below sequence diagram shows the initialisation and the signal watch operation sequence details.



Test Automation

Services of Automatic Test Framework may be harnessed in two ways namely, through a GUI and through an API set. The services are categorized in two different sets – A. Test Setup Editor and B. Test Suite Editor and Executor. The GUI modules of both of them employ the related helper modules or API sets. This is illustrated by the following component diagram:



Clearly, the client application interacts with the user interface modules that employ services of helper modules / libraries to serve the client application. The client application service interface is needed to carry out operations like frame transmission, interaction to virtual bus etc.

Below is a reiteration of the software's functions:

Test Setup Editor: This is the editor of a test setup file, with which it is possible to define and modify a number of test cases. The test setup editor UI is the front-end and realised as graphical user interface. Data manipulation is carried out by Test Setup Editor Library module and the artefact generated is the xml based test module file.

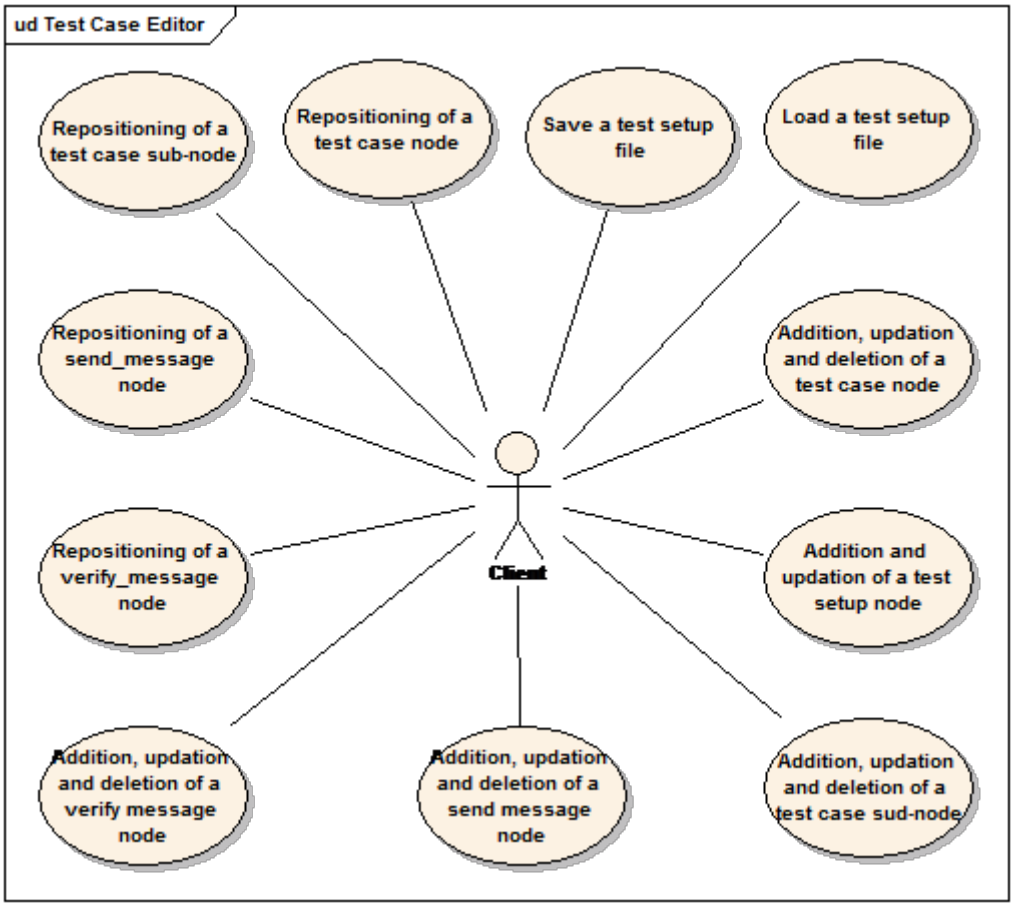
Test Suite Editor and Executor: Its purpose is twofold – the first one is defining and manipulating a test suite and secondly, executing the same by running the selected test setups and test cases. Output of the editing is a byte stream which the client application can store somewhere.

The Test Suite tree contains consistent types of leaves all being test setups that contains only test cases. Hence for Test Suite Editor and Executor Lib, just a set of API is sufficient. On the other hand, the Test Setup tree contains leaves of varying kinds going into varying hierarchy levels and given the functionalities like repositioning, it is best to define the Test Setup Editor Lib interface as a container class with all the other entities being inherited from a common class owing to their sharing of certain functionalities.

Because the UI truly reflects the data, test case editor lib and test setup editor & executor lib are having same use cases as the user interface modules.

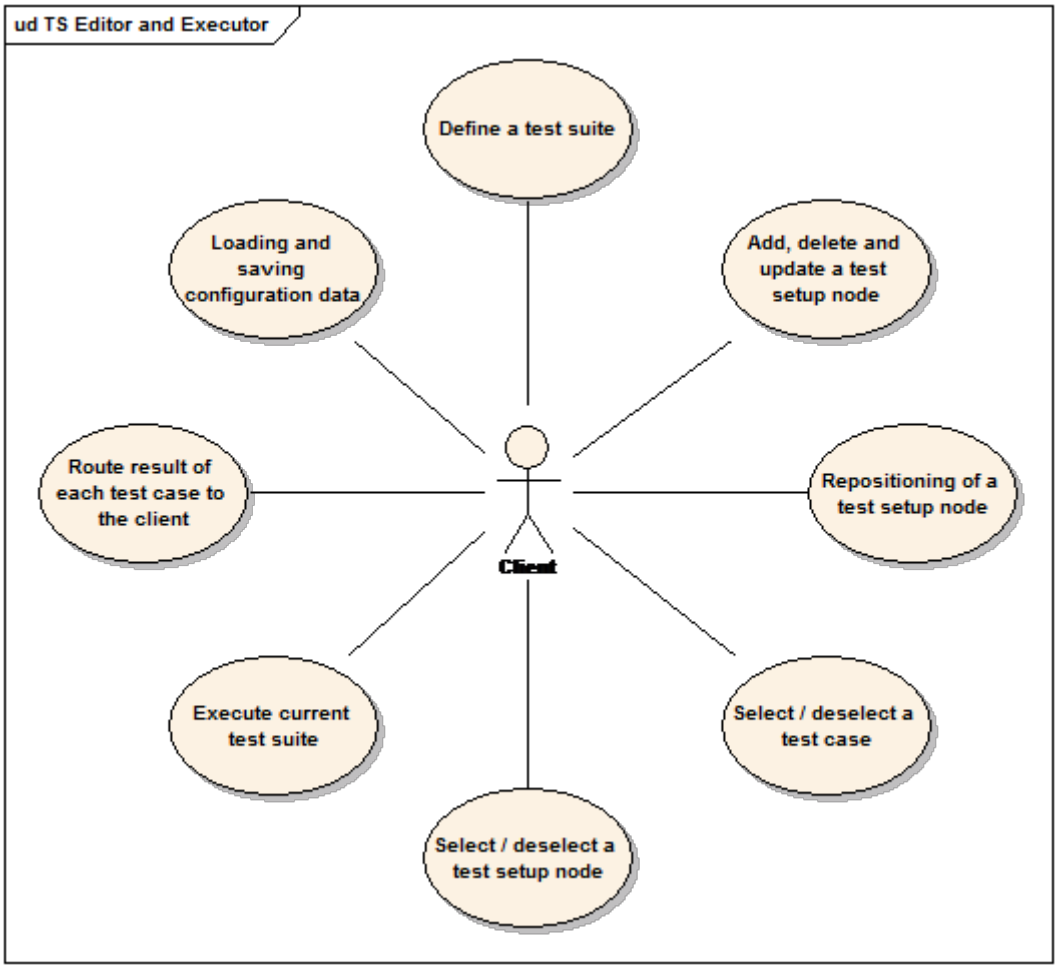
Test Setup Editor Lib module

This helper module is responsible for manipulating Test Setup file acting as an interface to the file and the Test Setup data repository. The services to be rendered by this module are first listed and then translated into an API set or a suitable class hierarchy.



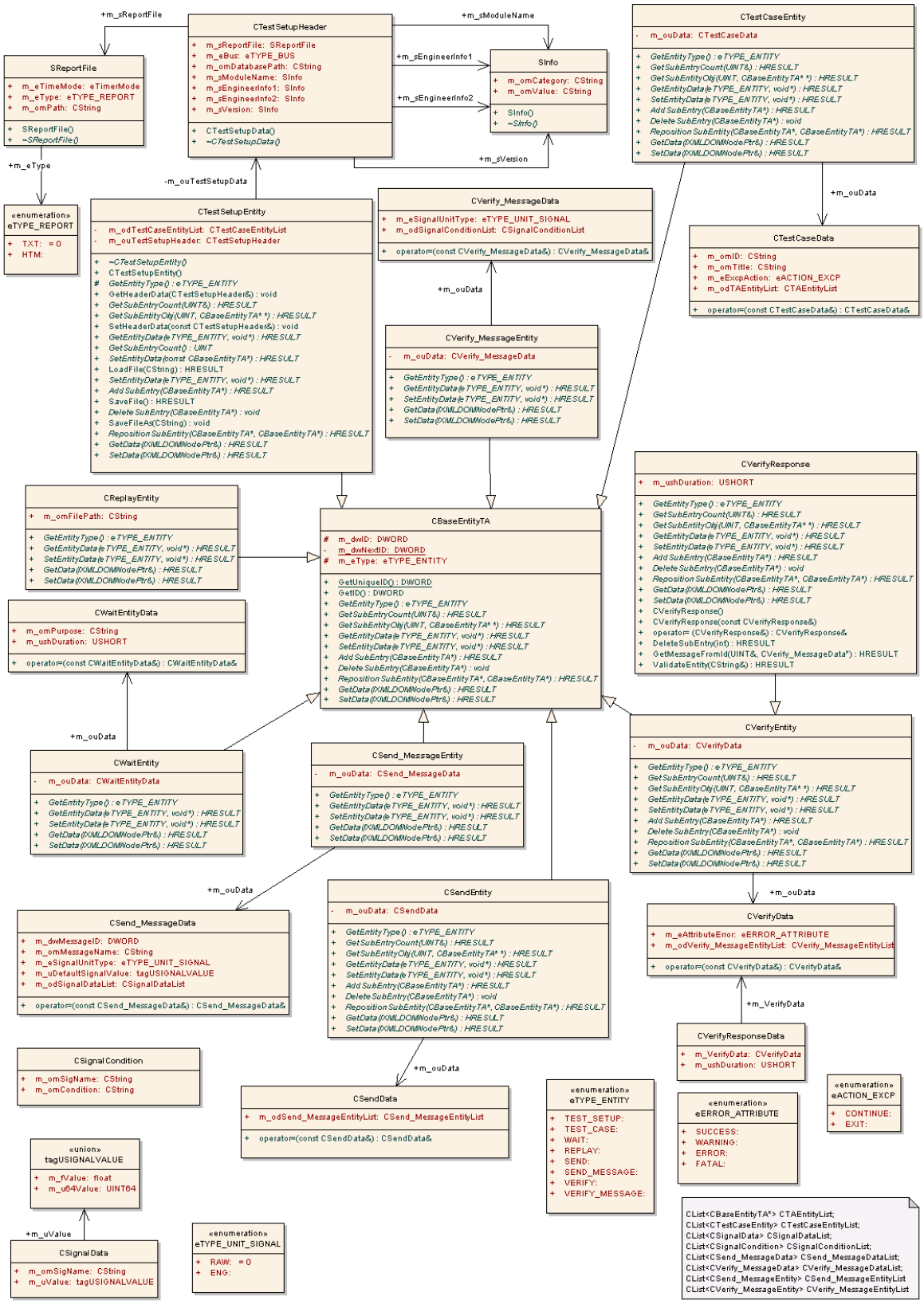
Test Suite Editor and Executor Lib

This helper module realises defining and updating of a test suite node along with its execution. Services provides by this module are listed down with a use case diagram.



Test Setup Editor

This helper module basically manipulates entities of varying kinds. For example – a test case node contains four different kinds of sub entities or nodes. Also, sub entities like send or verify also contain their own message sub-entities whereas nodes like wait, replay are simple nodes. Nevertheless, all of them must exhibit functionalities like addition, deletion, modification of self. Also, repositioning of a sub entity is also a desired functionality. Therefore, it is the obvious decision to inherit them from a common base class named as CBaseEntityTA (TA stands for Test Automation). Also, each of the objects has a data part. The overall class structure may be expressed by the following class design diagram where the Test Setup Editor Lib is realised by the class CTestSetupEntity



Automation Server Interface

The interface set has been defined in such a way that it renders the basic functionalities from BUSMASTER application suite. A concise view on the interface is provided below by the following class diagram.

IDispatch
IApplication
- Connect(BOOL) : HRESULT
- LoadAllDll() : HRESULT
- UnLoadAllDll() : HRESULT
- StartTxMsgBlock() : HRESULT
- StopTxMsgBlock() : HRESULT
- ResetHW() : HRESULT
- ResetSW() : HRESULT
- StartLogging() : HRESULT
- DisplayWindow(eWindow) : HRESULT
- GetMsgInfo(BSTR, sMESSAGESTRUCT*) : HRESULT
- SendKeyValue(UCHAR) : HRESULT
- LoadConfiguration(BSTR) : HRESULT
- ImportDatabase(BSTR) : HRESULT
- EnableDisableHandlers(BOOL, eHandlerType) : HRESULT
- GetErrorCounter(UCHAR*, UCHAR*, INT) : HRESULT
- SendCANMSg(CAN_MSGS*) : HRESULT
- GetNetworkStatistics(int, sBUSSTATISTICS_USR*) : HRESULT
- StopLogging() : HRESULT
- WriteToLogFile(USHORT, BSTR) : HRESULT
- SaveConfiguration() : HRESULT
- AddLoggingBlock(SLOGGINGBLOCK_USR*) : HRESULT
- SaveConfigurationAs(BSTR) : HRESULT
- AddTxBlock(STXBLOCK_USR*) : HRESULT
- GetTxBlockCount(USHORT*) : HRESULT
- GetTxBlock(USHORT, STXBLOCK_USR*) : HRESULT
- DeleteTxBlock(USHORT) : HRESULT
- ClearTxBlockList() : HRESULT
- AddMsgToTxBlock(USHORT, CAN_MSGS*) : HRESULT
- GetMsgCount(USHORT, USHORT*) : HRESULT
- GetMsgFromTxBlock(USHORT, USHORT, CAN_MSGS*) : HRESULT
- DeleteMsgFromTxBlock(USHORT, USHORT) : HRESULT
- ClearMsgList(USHORT) : HRESULT
- AddFilterScheme(BSTR, VARIANT_BOOL) : HRESULT
- GetFilterScheme(USHORT, BSTR, VARIANT_BOOL*) : HRESULT
- GetFilterSchCount(USHORT*) : HRESULT
- UpdateFilterSch(USHORT, SFILTER_USR*) : HRESULT
- GetFilterCountInSch(USHORT, USHORT*) : HRESULT
- GetFilterInFilterSch(USHORT, USHORT, SFILTER_USR*) : HRESULT
- DeleteFilterInSch(USHORT, USHORT) : HRESULT
- EnableFilterSch(EFILTERMODULE, BOOL) : HRESULT
- AddSimulatedSystem(BSTR) : HRESULT
- GetSimulatedSystemCount(USHORT*) : HRESULT
- GetSimulatedSystemName(USHORT, BSTR*) : HRESULT
- DeleteSimulatedSystem(USHORT) : HRESULT
- RemoveLoggingBlock(USHORT) : HRESULT
- GetLoggingBlockCount(USHORT*) : HRESULT
- ClearLoggingBlockList(void) : HRESULT
- GetLoggingBlock(USHORT, SLOGGINGBLOCK_USR*) : HRESULT

Interface Design

User Interface

•This section is applicable only if there is a human interface. Give reference to GUI standards document, if any. Mention about the GUI standards used as the basis for the design. Refer Guidelines on Database Design for more details.

<TBD>

Component Interface

Test Suite Editor and Executor Lib

Interface of this module is tabulated below:

Name & prototype	Functionality	Parameter details	Return values	Remarks
HRESULT SelectBus([in] eTYPE_BUS eCurrBus)	Sets the Current bus type for test suite.	eCurrBus - Current bus indicator	S_OK, ERR_NOT_IMPLEMENTED	
void SetTestsuiteName([in] CString omName)	Sets name of the test suite	omName - Intended name of the test suite.	-	
HRESULT AddTestSetup([in] CString omFilePath, [out] DWORD& dwID)	Adds a test setup	omFilePath - Path of the test setup file dwID - The unique ID of the test setup if the function is successful.	S_OK, ERR_PATH_INCORRECT, ERR_FILE_INCORRECT, ERR_ALREADY_ADDED, ERR_BUSTYPE_MISMATCH	
HRESULT UpdateTestSetup([in] DWORD dwID, [in] CString omFilePath)	Redefines an existing test setup	dwID - ID of the test setup. omFilePath - Path of the test setup file	S_OK, ERR_PATH_INCORRECT, ERR_FILE_INCORRECT, ERR_ALREADY_ADDED ERR_WRONG_ID	
HRESULT DeleteTestSetup([in] DWORD dwID)	Deletes an existing test setup	dwID - ID of the test setup.	S_OK, ERR_WRONG_ID	
HRESULT EnableTestSetup([in] DWORD dwID, [in] BOOL bEnable)	Enables / disables a test setup	dwID - ID of the test setup. bEnable - TRUE to enable, else FALSE	S_OK, ERR_WRONG_ID	
HRESULT RepositionTestSetup([in] DWORD dwID, [in] dwIDPreceding)	Repositions an existing test setup entry after another one.	dwID - ID of the test setup. dwIDPreceding - ID of the preceding test setup entry. If it is invalid (= ID_INVALID), the shifting entry shifts to the frontmost position.	S_OK, ERR_WRONG_ID, ERR_WRONG_ID_REF	

Name & prototype	Functionality	Parameter details	Return values	Remarks
HRESULT GetTestcaseCount([in] DWORD dwID, [out] UINT& unTotal)	Call to get total number of test cases occurring under a test setup.	dwID - ID of the test setup. unTotal - Total number of test cases	S_OK, ERR_WRONG_ID	
HRESULT GetTestCaseInfo([in] DWORD dwID, [in] UINT unIndex, [out] STestCaseInfo& sTCInfo)	Getter for a test case basic information like its name and selection status.	dwID - ID of the test setup. unIndex - Zero based index of the test case. sTCInfo - Test case information.	S_OK, ERR_WRONG_ID, ERR_WRONG_INDEX	typedef struct STestCaseInfo { Cstring m_omName; BOOL m_bEnabled; };
HRESULT EnableTestCase([in] DWORD dwID, [in] UINT unIndex, [in] BOOL bEnable)	Enables / disables a test case	dwID - ID of the test setup. unIndex - Zero based index of the test case. bEnable - TRUE to enable, else FALSE	S_OK, ERR_WRONG_ID, ERR_WRONG_INDEX	
HRESULT Execute([in] PFCALLBACKRESULTTC pfResultTC)	Executes the enabled test cases	pfResultTC - Address of the callback function defined in the caller which will be called as a result of each test case execution. This can be null.	S_OK, ???	Callback function prototype defined as: void (CALLBACK* PFCALLBACKRESULT) ([in] DWORD dwTestsetupID, [in] CResultTC& Result)

Design Alternatives

No alternative design consideration because the only one has been found to be the best suitable so far.

Design Feasibility

Design feasibility has been ascertained by theoretical analysis of the interface design of the two helper modules / libraries. Moreover, prior experience in somewhat similar data manipulation class design and implementation also contributes to the confidence.

Design Tools used

Enterprise Architect is used as the design tool

Additional Hardware and Software required

No additional hardware and software other than what is being used in BUSMASTER development, is necessary, at least to run in simulation mode. But in order to use it with the network, one of the supported controllers has to be procured.

Test Strategy

Both manual and automated testing is utilised. For the later AutoIt environment is utilised.

References

List of all external sources of information referenced in this document.

SI. No.	Description	Date	Vers.	Location

Description, date, and version shall uniquely identify the information source, and the location shall specify where it is to be found.

References to books or journals may be given using the following format: Author [s]: Journal Name / Book Name, publisher, date of publication.