Message Sequence Charts with Moritz and Doxygen

This tutorial describes how to create message sequence charts with Moritz and how to include them in the documentation created by Doxygen.

Please refer also the tutorial part that explains how to set up the used tools and the architecture of the example project as well as the provided sub-directories for Moritz and Doxygen.

Content
Message Sequence Charts with Moritz and Doxygen.................................................................1
  Common Explanations.............................................................................................................2
    The Moritz Configuration.................................................................................................2
    The Doxygen Configuration............................................................................................2
    Note .................................................................................................................................2
  What Message Sequence Charts are..................................................................................3
    Message Sequence Charts don't describe Opportunities................................................3
    Message Sequence Charts describe single Process Examples........................................6
  The Work-Flow and its Basic Conditions ..........................................................................7
  Stimulation and Simulation...............................................................................................9
    Representation of a Use-Case..........................................................................................9
    Test Extension................................................................................................................11
    Environment Simulation ...............................................................................................13
  Generation of Probed Sources.........................................................................................16
    The used Library............................................................................................................16
    Using Moritz as Code-Generator....................................................................................17
    The modified source .....................................................................................................18
  The IDE Probe Projects....................................................................................................20
    The Original Project.....................................................................................................20
    The Modified Project....................................................................................................22
  Generating Message Sequence Charts.............................................................................24
    Creation of MSC Files..................................................................................................24
    Including single Charts...............................................................................................24
  Resources and Links..........................................................................................................25
**Common Explanations**

Please read the Chapter “Run and Explore” in the tutorial introduction to learn how Moritz and Doxygen will be set up basically for all tutorials. This chapter explains only some points which may be important in this special case.

**The Moritz Configuration**

Since this tutorial shows only how to include message sequence charts into the doxygen output there will be now additional configuration for other diagram-types.

The configuration files used are:

- `ansi_c_abc2xml_cfg_prb.xml`:
  - configuration of abc2xml to parse the sources
  - different to the common configuration of abc2xml, Moritz commands will be treated as normal comments only
- `ansi_c_xml2abc_cfg_prb.xml`:
  - configuration of xml2abc to create a copy of the analyzed sources with inserted probe-commands which collect data while run-time
- `ansi_c_xml2abc_cfg_msc.xml`:
  - configuration of xml2abc to create message sequence charts from the data collected from the probe-commands inside the copied sources

Different to the generation of nassi shneiderman or UML like activity diagrams the generation of message sequence charts will not be done in one evaluation step. Thus the scripts mentioned above will be used in 2 independent runs. Furthermore Moritz commands will be ignored and treated as normal content of comments.

**The Doxygen Configuration**

Since Doxygen should use the Mscgen scripts created by Moritz the location where to find them has to be configured in the “Doxyfile”. This is done by adding the path-string to the configuration option “MSCFILE_DIRS” like this:

```
MSCFILE_DIRS       = ../msc
```

**Note**

- To update the content of the Doxygen output a complete update of the Moritz sequence has to be run first, if the diagrams should be updated.
- Since message sequence charts will be generated as Mscgen scripts and will be transferred into graphics by the Mscgen tool, this is needed to use them.
- Once the diagram images are created they can be used for all kinds of documentation that is able to contain graphics.
**What Message Sequence Charts are**

Even message sequence charts are well known today, it may be helpful to explain them, to have a common base and to understand how they will be created by using Moritz, since the process is more complex, than the generation of nassi shneiderman or UML like activity diagrams.

**Message Sequence Charts don't describe Opportunities**

The reason to generate diagrams for documenting software is usually to give the reader an additional view point to the written text by displaying relationships between code parts located at different locations. For different kinds of relationship there are different kind of diagrams available.

One important property of software is the set of algorithms used to solve specific tasks. This algorithms are usually realized as functions which contain commands to solve detail tasks and decisions to control, if a command sequence should be used or repeated. This means that a common function contains more commands than necessary, if it will be called only once. But it contains enough commands to do its job, if it is called an unlimited number of times.

Thus a diagram that displays a common algorithm displays potential opportunities, since every if-or switch-decision contains at least two decision paths where only one will be used at one time. Furtehermore many loop definitions allow different numbers of repetitions.

To display the algorithm, used in a function in a more compact way, the nassi shneiderman diagram (also called structogram) is used.

![Nassi Shneiderman Diagram](image)

Its box-like design makes it very easy to atomize its generation. Moritz is using nested html tables to do this. The columns in the diagram show that for one call only a subset of all contained commands is relay used and with every call of the function a decision process takes place to define which command or command sequence will be used for the specific call.
An other possibility to display the content of a function is a flow-chart or in his modern form an activity diagram as known from UML.

![Flowchart diagram]

It is less compact as a nassi shneider diagram and it takes more place but often it is easier to survey. Its disadvantage is that it is more difficulty to generate it automatically and the generator may place some elements in a confusing position, like in this case the activity “parkAtEnd” what may be better placed in one horizontal line with the other 3 activities. However, this kind of diagram shows all possibilities as well as the decisions to control what activity should be called.

A message sequence chart should not be used this way. Even some modern tools use them to display the whole algorithm of a function, as it will be done by a nassi shneiderman or an UML like activity diagram, it is not the original idea behind. That's why it is called message sequence chart since it is only showing the really used sequence of commands used while one single specific call of the function.
Usually one routine is not doing its work alone. Today many algorithms are to complex to be implemented as one single function and so one is calling others to delegate details. The called functions may do the same and so a call relationship is the result.

This call relationship will be shown by a caller graph as known from Doxygen.

![Caller Graph](image)

Even a caller graph shows not only one function but its relationship to others, it is still a description of possible opportunities. If the root-node is called this means not that every other node is used. Furthermore some used nodes may be used more than once while one call of the root.

Even a message sequence chart shows called functions also, it should not show all which may be called but only those really called, if the root is called once. On the other side if a called node is called more than once, it will be shown as often as it is called. Furthermore a message sequence chart displays exactly the order in which one function is calling other functions, that's why its name contains the term `sequence`.

So a message sequence chart displays not the opportunities a function has for different use-cases. It shows how it works for especially one use-case.
A message sequence chart displays how a specific example of an use-case will be processed as a contribution of several functions.

In this example the user is not only calling 1 function but 3. The first 2 calls are used to prepare the call of the third function. This means the stimulation of a use-case example means not only to call one service-routine alone. It means to do some preparations to define the whole use-case scenario. This can not be done automatically, since it has to be done by respecting specific requirements which are normally located in independent documents and the sources are only derived files.

Thus to generate a message sequence chart the function for what the diagram should be generated has to be used inside an additional code-frame that ensures the use-case depending environment in which the function should work and call other functions.

Driven by its given parameters as well as by the environment the function starts its work and as the result of specific decisions other functions will be called. This other functions may be located inside the same module where the root function is located. But most of them are located somewhere else and this functions call again the next. Loops inside a function result in sub-sequences which are a repetition of a sub-sequence of commands before with the same or differing parameters.

The whole process is a specific sequence of function calls where specific parameter values will be used. Thus a message sequence chart displays only the specific use of those opportunities which are really necessary for one specific use-case example.

Since the generation of a message sequence chart needs a stimulating code frame to configure the use case, this can also be used to check at the end some modified values, what results not only in stimulation of the root function but in a test of it. Thus a message sequence chart can also be generated while a test and to document its internal work-flow.
The Work-Flow and its Basic Conditions

The circumstance that a message sequence chart represents a specific example of an use-case makes it necessary to follow a work-flow for their source depending creation. This work-flow expects special conditions.

The name “message sequence” explains the idea of a kind of communication between independent but collaborating modules. Thus the work-flow expects the existing of at least 2 different source-files or or classes. In the used example the component contains 6 source-modules where every module contains a collection of specific functions and internal data-objects. But the component contains not the main program and it may be only one component under some more. The whole program contains at least one additional module with the main program, that is using the application interface of the module as a collection of tools. Sibling components have their own application interface to provide additional functionality. The whole program may be used to control an external object and thus the component has not only an application interface but an environment interface also. The environment interface is used from the component to transmit data to the environment and to receive data from it. This data-transfer is as important for component as the data exchanged with the application.

In normal projects it will be very difficulty to use the original environment and the original application for generating message sequence-charts and actually it is not really a good idea to do so even it is possible. Using replacements instead makes it possible to tread every component independently from the others and changes in one component may not influence the result of others.

To replace the application means to write a new one, that is only using the single component. Instead of the sibling components the program contains special stimulation or test functions with predefined data-sets to replace data from other components and perhaps with special checks to ensure that component to generate message sequence charts for really works as desired.

To replace the environment can be a very tricky task, since it may result in establishing a kind of dynamic simulation especially if the component expects plausible reactions for specific commands. Once a sibling component is used directly by the component to generate message sequence charts for its replacement is part of the environment simulation.
The work-flow to generate message sequence charts is a sequence of the following steps:

1. The stimulation of the component together with the simulation of its environment is the first step and even if it is not really needed, it will be always better to implement the stimulation of an specific use-case example as a complete and automated test to ensure that changes and unexpected results will be detected as soon as possible.

2. Since message sequences will not be generated from the source-code but from data collected while using the component in a specific scenario, probing functions have to be inserted in every function of every module that should be shown in the diagram.

3. To create the data collections used for the diagrams, the probed functions have to be used instead of the original not probed functions together with the stimulation sources and those used to simulate the environment to build a program independent from the real destination program.

4. The new program has to be started and its data-output has to be stored. Since this program may be used to run diverse specific use-case examples, it is very important to store the collected data in a way that allows to identify for every use-case example its own data.

5. Once the data created for every specific use-case example is available, it can be used to generate the associated message sequence chart from.

Using the collected data to generate a message sequence chart from is the goal of this tutorial. But actually it gives you some more possibilities. Usually use-cases of a component are only available for its external interfaces but not its internal especially if the internal design of the component is in the responsibility of the programmer. But every use-case example of the component interface is also a use-case of an internal module not seen from outside. Thus the data collected while running the specific use-case example contains the specific use-case examples of the internal modules also and this data may be useful for defining additional stimulation programs used to generate message sequence charts for internal modules.

Furthermore the collected data can be used to define tests. Once it turned out that the component is doing its job as desired (by additional tests, by long term use without issues, ...), the collected data can be used to detect unexpected behavior changes after an additional change of the source-code.
**Stimulation and Simulation**

The idea behind the terms stimulation and simulation is, that the application interface is used to insert specific data into the component and this data is than will be transferred directly or indirectly to the system-environment of the component. The system-environment replies with with a specific reaction and providing of data inserted into the environment interface of the component. The component transfers this reply-data back directly or indirectly to the application interface, where it can be readout again.

It may be, that very simple components may not need a complex simulation of the environment. That is especially the case, if a fire and forget philosophy is used. But since that is the simpler case this tutorial will show the more complex one.

**Representation of a Use-Case**

A real application may contain its own complexity wile using diverse other components. Furthermore the component may be not used directly from the main-program but over an other component.

How ever the real application may be designed, at the end it inserts data into the component and this data represent a specific use-case example for the component. But to generate message sequence charts for there is no need for the real application as long as all input-data is known.

Instead of the main-program a simple function can be used. This simple function provides all data for the specific run and inserts it into the component. This may be done by calling special set-functions of the component to initialize its internal variables. While doing this, it is essential to initialize all internal variables to avoid not wanted dependencies between 2 or more use-case scenarios. Since the component works together with its environment simulation, this has to be initialized completely as well.

Once the initialization is finished the a function of the component interface is called to start the use-case reaction that creates the data-collection used as base for the message sequence chart.

While the real application is using a component without inserted probes the data-collection will be done with modified sources, which call additional functions used as probes. To ensure that the modified sources do their job as well as the original sources, it is recommended to check some data-objects which allow to evaluate, if the original sources fulfill their requirements as well as the modified.

If one specific use-case example is represented simply by a function, several of this functions can be called in one main-program. But it has to be ensured, that the behavior of one function depends not on its position in the call-sequence. Therefore some effort has to be spent in the initialization of all data-objects and the definition of the checks after running the component.
This is the nassi shneiderman diagram of one example function that represents a specific use-case example.

In this example the initialization of the environment simulation will be done first by defining the content of a data-structure and the call of the associated initialization function. The adjustment of the component itself is done by using its normal interface functions and to see this as an independent initialization or as a part of running the component does not really matter as long as the right call order is ensured. After the last call of the component-interface the simulation interface is used again but now to evaluate, if the component has done its job as required.

While the component was working, its sequence data was collected and it will be stored in its own xml file before the associated memory will be freed again.
Test Extension

Once the stimulation should be extended to a complete Test, it will be an advantage to use a complete test-library like “UnitTest Cpp”, “Cunit”, CPPUnit, or another one. Diverse free test-libraries are existing and they support not only the design of single tests but their combination to complete test-programs also.

This tutorial should explain how to use Moritz to generate message sequence charts and the introduction in the use of a test-library needs normally its own tutorial. Thus in the example own test module is used, that provides one function to compare 2 integer values and one function to compare 2 float values. Both functions expect 2 numeric values to compare and an enumeration value, that explains what kind of comparison should be done. The comparison will be executed as a test, that may be passed or that may fail. In the case of a fail an error-message will be created.

Please note that for float values tests for equality and inequality are not supported. One reason is that the reference value of a test is usually a string-literal, what will be transformed into a float-value while run-time. In the normal case such a literal could not be transformed exactly, since the float-format is not able to represent every possible number. This is a limitation of the float format and not of the used software-algorithm. Instead of compare 2 float-values, it is the better choice to check, if the float-value is located inside a specific range defined by an acceptable tolerance.

For convenience reasons the error-messages have a special format, what can be displayed from special IDEs in their compiler-log window. If the start of the test-application is part of the build-process, the test outputs will be treated like compiler or linker messages. If the IDE has a link-feature a mouse click on the error-message results in a jump to the location, where the specific check was done.

The basic idea behind was originally taken from the test library “UnitTest++” formally available under “http://unittest-cpp.sourceforge.net/” and today moved to “https://github.com/unittest-cpp/unittest-cpp/” originally written by Noel Llopis and Charles Nicholson and today also maintained by Patrick Johnmeyer.

They show that depending on the kind of IDE the respecting of a specific syntax the output will be recognized from the IDE like the output of the compiler and it will be processed like it. Therefore it is essential to log not only the data used to define the content of the error-message but the name of the source-file and line-number associated with the error-message also.

This can be done by using the preprocessor constants __FILE__ and __LINE__, which are already known by a common ANSI-like build environment.

While the check-function used in this tutorial has the following prototype:

```c
CHECK_RESULT checkFloat(float value1, float value2, CHECK_TYPE check, const char* fileName, int lineNumber);
```

The test-function calls a macro, that is using the preprocessor constants __FILE__ and __LINE__ to collected the necessary file-positions automatically. Thus the user does not have take care about this and has only to provide the to numbers to compare and the kind of check:

```c
#define CHECK_FLOAT(value1,value2,check) checkFloat(value1, value2, check, __FILE__, __LINE__)
```
This are the code parts used to define the output-string format located in the stimulation-module “check”.

```c
#if defined(GNUG)
    char const* const LogStringInt1 = "%s:%d: check failed: %d %s %d\n";
    char const* const LogStringFlt1 = "%s:%d: check failed: %f %s %f\n";
    char const* const LogStringInt2 = "%s:%d: check impossible: %d %s %d\n";
    char const* const LogStringFlt2 = "%s:%d: check impossible: %f %s %f\n";
#else
    char const* const LogStringInt1 = "%s(%d): check failed: %d %s %d\n";
    char const* const LogStringFlt1 = "%s(%d): check failed: %f %s %f\n";
    char const* const LogStringInt2 = "%s(%d): check impossible: %d %s %d\n";
    char const* const LogStringFlt2 = "%s(%d): check impossible: %f %s %f\n";
#endif

printf(LogStringXYZ, fileName, lineNumber, value1, "??", value2);
```

Controlled by the preprocessor constant “GNUG”, that has to be provided as an argument of the compiler a package of character constants will be defined. If it will be not provided, other character constants will be defined instead. While in both cases every string starts with the file-name and the line-number the detail shape differs a little bit. In the case of “GNUG” as separator between file-name and line-number a “:” is used. In the other case the line number is placed between “(” and “)”. In both cases there is no space used in between and the combination of file-name and line-number ends with a “:”. Behind that the error message follows and even its format is more or less free it is a good idea to think about its shape to have a usable output.

Even the basic idea is taken from a specific test-library, the tutorial uses an independent solution and it is not necessary to have an additional test-library. Never the less it is useful to have one for a real project. But it is not the goal of Moritz to support a special one. It should be possible to use Moritz with diverse test libraries already existing.
Environment Simulation

The simulation of the environment depends on the task the component is responsible for. In the used example the component is using 2 digital output terminals to accelerate a vehicle forwards or backwards or to stop this movement depending on the current position. Furthermore a voltage input terminal and two digital input terminals are used to transfer feed back information into the component.

In a first step the simulation has to read the current content of the component outputs to evaluate the desired direction of the movement.

Depending on the desired direction and and the desired acceleration or deceleration a resulting speed will be calculated. As long as an active movement is desired the speed in the desired direction will be incremented by a constant value until a maximum speed is reached. In the case the movement should be stopped the speed-value will be decremented by a different constant value until the movement stops. While one call of the simulation the speed will be changed only one times.

If the current speed is known, the new position will be calculated. It will be assumed that between 2 calls of the simulation a constant time-step passes by but as well as the position the time has no real unit.

If the position is known, the component inputs can be calculated. By using a simple characteristic curve with an offset and a gradient a sensor voltage will be calculated that represents the position. This voltage will be used to calculate the content of adc register used as analogue input-data. Parallel to that the content of 2 digital input ports will be defined, used from the component to detect that the forward-limitation or backward-limitation is reached.
While the calculation algorithms are located inside some functions of the simulation, all used simulation parameters are located in 2 structures one for the mechanical data and one for the electronic data. The content of this structures can be changed by using initialization functions.

Furthermore it is possible to readout a data structure that contains the current calculated data. This makes it possible to check if the component fulfills the requirements associated with controlling its environment.

Having a simulation is one thing. But how will it be called without a kind of task management with independent threads?

One possibility is to call the simulation in the component for what the simulation should be used.

```plaintext
void placeAt ( int position )
int CurrentPosition = 0
static int OldPosition = 0
int RemainingSteps = 20
int Tolerance = 1
CurrentPosition = getPosition ( 1 )
print( "initial position: %d \n", CurrentPosition )
while ( ( RemainingSteps > 0 ) && ( abs_int ( CurrentPosition - position ) > Tolerance ) )
do a limited number of times until position inside tolerance

OldPosition = CurrentPosition
CurrentPosition = getPosition ( 1 )
print( "current position: %d \n", CurrentPosition )

if ( ( position - CurrentPosition ) > 0 )
    if ( ( CurrentPosition - OldPosition ) < ( ( position - CurrentPosition ) / 2 ) )
        moveForwards ( 1 )
        TRIGGER_SIMULATION ( "moving forwards \n"
    else
        stop ( 1 )
        TRIGGER_SIMULATION ( "stop \n"
else if ( ( position - CurrentPosition ) < 0 )
    if ( ( OldPosition - CurrentPosition ) < ( ( CurrentPosition - position ) / 2 ) )
        moveBackwards ( 1 )
        TRIGGER_SIMULATION ( "moving backwards \n"
else
    stop ( 1 )
    TRIGGER_SIMULATION ( "stop \n"

print( "end position: %d \n", CurrentPosition )
```

This is one function of the tutorial example and it is part of the module called regulator. It is not an interface function but a private one of the module called by the function “moveTo”. Its core is a while-loop that contains a nested if-decision. At the end of each decision-path a command called “TRIGGER_SIMULATION” is used to call the environment simulation.

This means in the tutorial the component is responsible to call the simulation. But how it could be avoided, that the simulation will be called in the real application where the real environment should be controlled and not its replacement?
The command itself is not a part of the simulation but its definition is located in the same source file than its calling functions. Furthermore it is not a function but preprocessor macro.

The basic idea behind is have a command that implements 2 different behaviors. Controlled by an external element only one of both is used.

If the preprocessor constant “TEST” is defined, the header of the simulation module “simple_move” will be used and the macro “TRIGGER_SIMULATION” will be defined as an active code snippet. First the simulation will be really triggered by calling the function “simple_move_calculate” what results in the sequence explained above. For a better understanding what is going on some user-output will be created.

If the preprocessor constant “TEST” is undefined the macro “TRIGGER_SIMULATION” will be defined as an empty preprocessor constant that contains no action.

In C/C++ a preprocessor macro will be replaced by its content before the compiler translates the source-code into the binary code. Thus once “TEST” is defined, the triggering of the simulation is part of the component. Otherwise as long as “TEST” is undefined, the simulation will not be called. Using the preprocessor makes it easy to avoid adding the simulation into the application, if all parts which know the simulation are located inside compiler-switches controlled by “TEST” since once the compiler is not translating its trigger function, the linker is not missing its object code.

In other programming languages this may be more difficult. Instead of a not possible preprocessor construction a real function has to be used. And once the simulation should not be used the content of the function has to be commented out. But in the next versions it should be one feature of Moritz to support an automated way to do this.

Since Moritz generates for all component sources manipulated versions, which contain probe-commands the idea is to have special comments, which will be transferred into active code while generating the modified sources. How ever for C/C++ this is not necessary and examples for other programming languages will follow if a more or less stable functionality of this new feature is reached.
**Generation of Probed Sources**

While stimulation and simulation are used to represent the use-cases and the environment of the component, the data-collection used as base to generate message sequence charts has to be created by the component itself. But actually this is not part of its real functionality. Thus a modified version is necessary, that contains additional code lines used like measurement probes.

**The used Library**

The probe-commands are implemented in a library provided by Moritz. It can be found in the “LangPack” folder of the used programming language in its sub-folder “x2a_seq/lib”. Since this feature is still in its early development-phase and some details have to be still clarified only ANSI C/C++ is currently supported. Other programming languages will be supported later.

For convenience reasons the folder “lib” is copied into the folder “src” of the tutorial.

The library for C/C++ contains currently 4 modules:

1. **Sequence:**
   - contains a core class responsible for the real work
   - written with C++ language parts not compile able with a C compiler

2. **SequenceC:**
   - contains the used probe-commands
   - provides functions interface that can be compiled by a C compiler also
   - is using internally the core-class provided by the module above

3. **SequenceU:**
   - contains an additional log function for non standard types
   - has to be adapted by the user, if the component contains self defined data types, classes or other complex data-objects, which result in a wrong collection of data and wrong display of arguments in the message sequence charts

4. **_Tool:**
   - contains some auxiliary functions used by the sequence modules
   - originally developed for the binaries abc2xml and xml2abc and reused here

The library provides most of the standard probe-commands used inside the observed function. Only those probe-commands used to observe entering or leaving a function, are not provided by the library, since they will be generated individually for each function. The probe-commands are functions, which use internally logging commands provided by the library. For C standard types this is completely implemented in the library. But for own data-types created with `typedef` as derivatives of standard types, as structures, enumerations or unions and for classes, pointers, iterators and everything else not logged correctly in the view of the user, a user log-function is located in the module “SequenceU” that is used by the probe-commands and can be extended or modified by the user.

The logged data will be stored in a data-collection that is organized as tree. Entering a decision or loop inside a component function or calling an other function results into a new tree-node. Finally this tree will be stored from the library as xml file.
Using Moritz as Code-Generator

Having a library that provides all needed logging-functions is one thing. But to add this commands into all functions in all sources of the component is something else especially, if this should not be done in the original sources but in a copy and the component itself is still under development. Doing this manually will not be a realistic task for developer or tester.

Since the core job of Moritz is the generation of text-files why not use Moritz for generating the manipulated component files? Is this possible? It is. Moritz is not only able to generate dot-scripts or html-files. Once the binary abc2xml has transferred the sources-texts into an xml tree, the binary xml2abc is able to extend this xml-tree for inserting the probe-commands. Therefore specific configuration files are used:

- **ansi_c_abc2xml_cfg_prb.xml**:
  - configuration of abc2xml to parse the sources
  - different to the common configuration moritz commands will be treated as normal comments only

- **ansi_c_xml2abc_cfg_prb.xml**:
  - configuration of xml2abc to create a copy of the analyzed sources with inserted probe-commands which collect data while run-time

The created xml files differ a little bit from those created for generating nassi shneiderman diagrams or UML like activity diagrams, since Moritz commands used inside the source-code for controlling the diagram style should not influence the code-generation. Thus the evaluation of source-comments will skip analyzing Moritz commands and they will be treated as normal parts of the comments. The new xml files will be stored in the folder “xml_prb”.

Once the new xml files are available, xml2abc starts its work. Every file will be read. In the new distribution of Moritz the LangPack folder of the associated programming language contains a new sub-folder “x2a_seq” that contains not only the lib-folder of the sequence-library but x2a files used to configure xml2abc for doing its job also. Once an xml file is loaded, it will be extended. Above every function node 2 additional function-nodes will be inserted, which describe the probe functions used to log data immediately after entering the function and immediately before leaving it. This are very specific commands, since the parameter-interface and return-values differ from function to function. The modified sources will contain above every original function the definition of this probe commands. Inside the function-bodies the call of the probe-commands will be inserted. This are not only the probe-commands to log entering and leaving the function but probe-commands entering and leaving internal decisions and loops also.

Now the extended xml file contains all necessary information needed to generate the modified source file. Normally the modification of the xml file will not be stored. But actually this is not necessary for the end user since the modified versions of the component files are the important result.

The whole process works like creating diagrams. The call of abc2xml and xml2abc will be done by a terminal-script or batch-file called “create_Prb”. But instead of diagrams the result are source files. The output-files are located in the folder “prb” and its sub-folders. Moritz tries to use the original folder-structure with sub-paths relative to the location of the calling terminal-script or batch-file. Keeping the folder structure of the original-sources should make it easier to create for the IDE an additional source-project by copying and modifying the one for the original sources.
The modified source

The modified sources contain the same functions as the original ones. But their bodies contain additional command lines. Two of this commands are very specific and differ from function to function. Thus the definition of this specific commands is inserted in the modified source file also.

As an example this should be shown for one function called “moveTo”

Over the function itself, the definition of its specific log-function is done. For the compiler this works as declaration of the prototype also. Thus it is not necessary to insert the prototypes in the associated header-file, if the programming language C/C++ is used.

The first specific log-function is the probe-command used to store the data known while entering the component function.

```c
///
/// log function called inside replacement of:
/// void moveTo( int position)
/// used to log parameters while entering the function
///
///
void PRB_moveTo_ENTER(
    char * fileName,
    int lineNumber,
    int columnNumber,
    int position)
{
    Probe_beginOfFunctionCall(
        "moveTo",
        fileName, lineNumber, columnNumber);
    Probe_log_Object("int", "position", (void*)(&position), DataProbe_Parameter);
}
```

For storing source-depending data the sequence library provides a command to log the name of the function, the name of the source file and line and column numbers, where the function definition starts in the original source file.

For storing the parameters an other command is provided by the sequence library is used. While the data type has to be given as string, the value will be given indirectly as void-pointer. As long as the type is a known standard type, the the command dereferences the pointer and casts the type. But once this is not possible, the command calls a second one defines in the module “SequenceU”. This second function has to be extended by the user to define how a non standard type should be logged. In addition the parameter name is stored together with a key word, since this function is multipurpose log-function.

The second specific log-function is the probe-command used to store the data known while leaving the component function.

```c
///
/// log function called inside replacement of:
/// void moveTo( int position)
/// used to log parameters while leaving the function
///
///
void PRB_moveTo_LEAVE(
    char * fileName,
    int lineNumber,
    int columnNumber)
{
    Probe_endOfFunctionCall(fileName, lineNumber, columnNumber);
}
```

For functions with a return-type other than void the same command provided by the sequence library is used as for storing parameters. But since in this case the original component function is a void function the associated command line in the probe-body is empty.

For storing source-depending data the sequence library provides a second command to log the name of the source file and line and column numbers, where the function definition ends in the original source file.
The component function itself in its manipulated form contains as extensions the call of “PRB” functions which are the probe-commands.

```c
void moveTo(int position)
{
    PRB_moveTo_ENTER("regulator.c", 57, 1, position); // log the entering of this function and its parameters
    PRB_SWITCH_ENTER("regulator.c", 59, 2, "position"); // log the leaving of a switch statement
    switch(position)
    {
    case PositionBeginn:
    {
        parkAtBegin();
    } break;
    case PositionEnd:
    {
        parkAtEnd();
    } break;
    default:
    {
        PRB_IF_ENTER("regulator.c", 71, 4, "\((position>=PositionMin)\&\&(position<=PositionMax)\)"; // log the leaving of an if statement
        if((position>=PositionMin)\&\&(position<=PositionMax))
        {
            placeAt(position);
        }
        else
        {
            printf("position %d is out of bounds\n", position);
        }
    PRB_IF_LEAVE("regulator.c", 71, 4); // log the leaving of an if statement
    } break;
    }
    PRB_SWITCH_LEAVE("regulator.c", 59); // log the leaving of a switch statement
    PRB_moveTo_LEAVE("regulator.c", 57); // log the leaving of this function
}
```

Probe-commands will be called, when the function is entered or left to store parameters and return-values, which should be used in the message sequence charts. In addition structure defining source elements like decisions and loops have their own probe-commands used to log the entering and leaving of this function parts. Most of the logged data is already defined as string or constant value, since it was already known while creating the modified source file. This is the case for the source name the line and column numbers and identifier-names. Expressions used in decisions and loops are stored as strings also. Only parameter values are not known while creating the modified sources, since they differ depending on the use-case example of the stimulation scenario.

In the practice some comments may differ in their line-structure a little bit. If the original comments are split into several lines, they may be inserted as one line only. This is currently a limitation of Moritz. But since the modification of the component files should not be used for a kind of code review, this should be no real problem as long as the behavior of the resulting binary differs not from using the original source except the addition logging of data.
The IDE Probe Projects

While in the past very simple text editors where used and the build-tools where started manually, today integrated development environment are more or less a standard. But since the configuration of such a tool differs from IDE to IDE, it is very difficulty to explain this for all. Thus it should be explained what has to be configured and not how. The user of a specific IDEs has to knows how to configure it. Never the less the shown examples are taken from the free open-source IDE called Code::Blocks.

The Original Project

The project with the original component is used to setup tests reused in the project with the modified sources to ensure that they still work in the same way as the originals. This is very important to avoid that a bug in the code-generator will be not recognized.

The IDE project used for the tutorial contains 3 main directories on the hard-disk.

1. The folder Doxygen contains text-files which are used to provide the definitions of related pages of the documentation.
2. The folder lib contains the sequence library as described above.
3. The folder src contains the original component with its interface, service and terminal modules as well as the stimulation and simulation as described above.

The original src directory contains the header-files with their function prototypes as well as the c-files which are used for the function-definitions.

In the example for the tutorial c and h file of one module are located in the same folder. But this is not necessary and special folders, which contain h files or c files only, are also possible.
Around the component some more directories are part of the project:

- As already mentioned the Doxygen folder is used for additional text-files which contain Doxygen c-comments to define related pages for the Doxygen output.
- The lib-folder contains the already described sequence library. Even the probe-commands are not used in the original sources the commands to prepare and save the data-collection are already part of the test-application. For the tutorial it was copied into the tutorial locations. But normally it will be provided by the Moritz distribution.
- Since the component is not able to work for its own, the simulation of its environment and the stimulation of specific use-case examples are necessary also.

1. The simulation contains a module that describes the controller periphery with its special function registers. In other projects the simulation of dynamic processes inside the peripheries like pwm generators or timers may be necessary also. For the tutorial this was done together with the simulation of the electronic and mechanical system beyond the controller in the module “simple_move”.

2. The stimulation contains the main-program that calls test-functions, which contain the initialization describing the specific use-case examples and the check of results.

To compile the files some preprocessor constants have to be defined inside the IDE to be used as compiler arguments:

1. Command used for create directories (please refer the C/C++ documentation about system() and mkdir() for details)
   - SYSTEM_FOR_FOLDERCREATE (the “system” command has to be used)
   - NO_SYSTEM_FOR_FOLDERCREATE (an own command “mkdir” is available)

2. Operation-System (define nothing for others)
   - WINDOWS
   - LINUX
   - not defined

3. Expected format of compiler-errors (please refer the sub chapter “Test Extension” under Stimulation and Simulation for details and define nothing, if “GNU” does not work)
   - GNU
   - not defined

4. Use test-application as main-program (define nothing for a normal main-program)
   - TEST
   - not defined
The Modified Project

The project with the modified component is reusing the tests together with the modified sources to create the data-collections, which are the base for creating the message sequence charts. On the one side the test-application starts the stimulation of the specific use-case examples. On the other side the checks at the end ensure that the modified functions are still working as desired.

The basic folder structure of the modified project is nearly the same as used for the original one. But it contains one additional sub-folder called prb.

The prb directory is the root of a sub folder system, that has the same structure as the src directory and it contains the modified sources.

The main difference is, that the content of the prb directory is the replacement of the original c files.

The original h-files are still in use. For the tutorial only the c-sources where used to create modified files from, since the h files contain only function prototypes but no function-bodies.
As well as the original h files are used in the modified project the lib folder simulation and stimulation are reused also.

While the sequence library was used in the original project only, since the commands to initialize and save the data-collection were parts of the test-application, now in addition the probe-commands are in use to fill the data-collection with specific content.

The simulation and the stimulation is used for the same purpose as in the original project.

To compile the files, the same preprocessor constants have to be defined inside the IDE to be used as compiler arguments, like it was the case in the original project, and this with the same values.

A compare of the original project and the modified one shows that the only differences are the used c-files, which are located in the prb directories. As long as the IDE project file is a text-file, the modification can be done by a text editor based on a copy of the original project file. By replacing the directory path-strings of the c files in the project-file the modification is easily done.

Once the modified IDE project is configured, the modified application can be build and started. A IDE like Code::Blocks allows to start the binary immediately after building as post-build step. Terminal-outputs will be redirected to the IDE and analyzed, if they contain compiler like outputs. As long as no real compiler-error and no check-error occurs the test-application starts use-case example after use-case example and stores its data-collection as an xml file in the folder “seq”
Generating Message Sequence Charts

Once the data-collections are created and saved by using the modified component as xml files the generation of the message sequence charts with Moritz and Doxygen can be started.

Creation of MSC Files

The Moritz configuration file used is:

- **ansi_c_xml2abc_cfg_msc.xml**: configuration of xml2abc to create message sequence charts from the data collected from the probe-commands inside the copied sources.

Since the xml sources are already existing the use of abc2xml is not necessary only the Moritz binary xml2abc. It is called by the terminal-script “create_Msc”.

The detail configuration, used to create message sequence charts, will be provided with the Moritz distribution in its LangPack folder for the associated programming language in its sub folder “x2a_seq”.

The result will be saved as msc files in the folder “msc”. This files are scripts used by the tool Mscgen to create the real images. Mscgen provided by Michael McTernan is used by Doxygen internaly to transform the msc scripts into images.

Including single Charts

All message sequence charts generated by Moritz are located in the sub folder “msc”. For every specific use-case example there is one file that name was defined in the associated stimulating function.

For including msc files Doxygen provides the command “**mscfile**” that expects the file name as parameter. It is possible to extend the file name with a folder path string. But the more convenient way is to add the folder path to the list of example-paths (MSCFILE_DIRS) in the “Doxyfile”.

In the tutorial example this command is added in the h-file that contains the prototypes of the test-functions used to stimulate the use-case examples.

```c
@ingroup STIMULATION
first test of the function @ref moveTo

@htmlinclude test_moveTo_c_F_test_moveTo_1.html

@mscfile test_moveTo_1.msc

void test_moveTo_1(void);
```

Even a message sequence chart documents the stimulated function of the tested component it is better to display in the documentation of the used use-case stimulation. On the one side the normal user of the Doxygen output expects for a function its interface description only and an overload with too many images may disturb him. On the other side the documentation of an use-case is actually the documentation of the component-interface especially if not only one interface function was used in one specific example.
Resources and Links

For downloading the mentioned tools and further information about them please refer the following links.

1. Moritz: distribution of the script generator
   - [http://sourceforge.net/projects/Moritz/](http://sourceforge.net/projects/Moritz/)
2. MuLanPa: project used to develop the grammar and parser for Moritz
   - [http://sourceforge.net/projects/mulanpa/](http://sourceforge.net/projects/mulanpa/)
3. Doxygen: tool to document software source
4. Graphviz: collection of diverse graphic tools
5. Mscgen: tool to generate message sequence charts
   - [http://www.mcternan.me.uk/mscgen](http://www.mcternan.me.uk/mscgen)
6. Code::Blocks: IDE for text based software projects