NON PROTEGE

Athena V5.0

User manual

DOMAINE

Simulation

Released

Direction Générale Technique

Edition	*Date*	*Issue*	*Rédacteur*	*Visa Rédacteur*	*Visa Approbateur*
Origin	08/02/2011	1	S.Millet		
Last Edition	01/02/2008	1	S. Millet		

TABLE OF CONTENTS

1. PREAMBLE

1.1 Document issues

Date	Issue	Author	Updating Purpose
08/02/2011	1	S.Millet	Document creation from AthenaV4 User Manual (DGT 10543)

1.2 Summary

This document describes the use of the V5 version of Athéna. This version uses the Eclipse framework and more precisely, the XText and Acceleo technologies.

1.3 Reference documents

1 Athena GForge website:

http://gforge.dassault-avion.fr/projects/athena/

2 Athena User Manual V3.0:

http://gforge.dassault-avion.fr/docman/view.php/9/653/AthenaV3.0-UserGuide-Issue-2-0.doc

3 Athena User Manual V4.0:

http://gforge.dassault-avion.fr/docman/view.php/9/824/AthenaV4-UserManual.doc

2. PURPOSE OF THE DOCUMENT AND PRESENTATION

The purpose of this document is to present the use and architecture of Athéna in its 5.0 release and specifically:

- its description language;
- its use for describing a simulation;
- the associated generators.

Athéna v5.0 allows the generation of a simulation code from a description formalised in the Athena Description Language (ADLv5):

- This ADL description follows mostly the rules of the previous versions but some aspects have been reviewed to allow a more complete description of simulations and of data needed for generating an executable simulation.
- The code generated is a monolithic C++ application which can encapsulate C, C++ or MATLAB User models.

This document also presents the API Model generator which complements the Athena generator. API Model allows the specification of a library interface and the generation of the interface source code.

3. SIMULATION AND DESCRIPTION LANGUAGE

This section presents what we call a simulation and the language used to describe it (ADLv5).

3.1 Simulation

3.1.1 Time management

ADLv5 implicitly describes a cyclic view of an environment where all computations are done using the previous state of all manipulated objects. It means that every generated simulation has to use a cyclic simulation engine where, at step *i*, all computations are done using step *i-1* values.

3.1.2 Models

We call model a set of functions with a given interface allowing the representation of a particular physical phenomenon or of particular system behaviour. The models design and development is not in the scope of the tools we describe here.

3.2 Simulation overall description

In the following, we list all the elements needed to describe a simulation:

- Implementation Language declarations
- Data type declarations
- Function declarations
- Processing declarations
- Architecture definitions
    - Architecture components
    - Architecture root
- Simulation specific definitions

3.3 Implementation language declarations

The general format for declaring a language is:

```
language <name>
```

Where:

- **name**: is the language identifier. This identifier is then used in the basic type and function definitions.

3.4 Data types declarations

The types' description view allows describing all the data types used in a simulation. More specifically, it allows the description of:

- Basic types: float, integer, boolean, string, etc.
- Vector types: container of a specific type of data.
- Map types: Hash table with a specific key type and specific value type.
- Struct types: Aggregation of data of any type.
- Union types: Container which can contain one type out of a set of types.
- Aliases: alias on another type.
- Validity functions: functions allowing testing the validity of instances of a specific data type.

A more complete description of each type follows.

3.4.1 Basic Type

The general format for description of a basic type is :

```
basic <name> begin
      language <lang_name_1>  : <translation>
      include  <lang_name_1>  : <includeName>
```

```
|     language <lang_name_n> : <translation> = <init_string>
end
```

Where:

- **name**: is the type name.
- **lang_name_1** : is the language, CPP for C++, MATLAB for Matlab.
- **init_string** : is the initial value of an instance of this parameter type.
- **translation** : is the generated string in the targeted language.

3.4.2 Array and Collection Type

The general format for description of array or collection of basic parameters is :

```
array < basictypename > name[I, J, K...]
vector < basictypename > <name>
map < basickeytype, basicvaluetype > <name>
```

Where:

- **name**: is the type name.
- **basictypename, basickeytype, basicvaluetype:** is the type of contained values.
- **I,J,K… :** are the dimensions of the array, they can be fixed or dynamic.

3.4.3 Structured Type

The general format for description of a structured parameter type is :

```
struct <name> [ <I,J,…> ] begin
      <type_name_1> <param_name_1> [ = "<value>" ]
      …
      <type_name_N> <param_name_N>
end
```

Where:

- **name**: is the type name.
- **I,J,..** : dimensions to be applied to dynamic arrays attributes.
- **type_name_1** : is the basic or structured type of a parameter included in this structure.

- **Value**: if the field as a basic type then an initial value can be specified

3.4.4 Union Type

The general format for description of a structured parameter type is :

```
union <name> begin
      <type_name_1>,
      …
      <type_name_N>
end
```

Where:

- **name**: is the type name.
- **I,J,..**: dimensions to be applied to dynamic arrays attributes
- **type_name_1** : is the basic or structured type of a parameter included in this structure.

3.4.5 Alias Type

The general format for description of a “define “type is :

```
define <alias> is <typename>
```

Where:

- **typename**: is the original type name.
- **alias**: is another name for the original type.

The interest of creating aliases reside into the possibility to associate different validity functions.

3.4.6 Validity functions

The general format for description of a validity function is :

```
validity <typename> <functionname>
```

Where:

- **typename**: is the type name.

- **functionname**: is the function name.

The generated code provides the skeleton of a testing function. The user must then complete this skeleton.

3.4.7 Exemples

```
basic Double begin
      language CPP : "double" = "0."
end

define Azimut is Double;

basic Long begin
      language CPP : "long int" = "0
end

array <Double> DoubleArray [1,2]
array <Double> DynamicDoubleArray [X,Y]

validity Long isLong

vector<Long> LongVector
map<Long,Double> MapExemple

struct Coordinates begin
      Double lat
      Double lon
      Double alt
end

union MyDoubleArray begin
      DoubleArray,
      DynamicDoubleArray
end
```

3.5 Function declarations

It allows the description of all functions used in a simulation. The provider of the function is not taken into account in this view (see the Mapping view). It contains, for each described function:

- The function name
- The function implementation language
- The function signature, that is a list of argument definition with for each:
  - The type: one of those described in the types' description,
  - The "modifier" of the argument: is it "in", "inout" or "out " ?
  - A name.

The general format for description of a function is:

```
function <name> <<language>> (<modifier> <type> <param1> ..
                      <modifier> <type> <paramN>)
```

Where:

- **name** is the name of the interaction. This name must be unique for the current context (prototype in which this interaction is declared);
- **language**: tells in which language this function is implemented, the language must have been previously declared .
- **modifier** : IN, OUT or INOUT.
- **type** : type of an argument.
- **param1, paramN** are the names of parameters, used as arguments of the function. This list can be empty;

3.6 Processing declarations

3.6.1 Definitions

The ADL allows the description of simple "procedures". These "procedures" consist in sequences of function calls or other "procedure" calls, calls which can be encapsulated in control structures. In term of use, there is no difference between a function and a processing. This section describes the elements of a processing:

- the declaration part,
- the description of local variables;
- the description of the call to a user function or a sub-processing;
- the control structures which can be used.

For each, we give below the vocabulary and grammar used, with comments and examples.

3.6.2 Declaration

A first description:.

```
processing <name> ( <modifier> <type> <argname> ...) begin

      parameter < local parameter declaration>
      event < local event declaration >
      …
      execute < function call >
      …
end
```

Where:

- **name:** is the name of the processing name;
- **modifier** : IN, OUT or INOUT.
- **type** : type of an argument.
- **param1, paramN** are the names of parameters, used as arguments of the function. This list can be empty;

3.6.3 Local variable

A local variable can be a parameter or an event. It will be described as follows

```
        parameter <type> <name> [ = "<value>" ]
        event <name> { <cond1>, <cond2>, … }
```

A "parameter" is a variable declaration. It consists in:

- **type:** the type of the variable;
- **name:** is the name of the processing name;
- **value:** initial value for basic parameters.

An event is a list of conditional expressions. Each must be true for the event to be raised. An event is evaluated when it is used, not when declared ! It consist in:

- **name:** is the name of the processing name;
- **condN:** conditions. A conditition is: <param1> <operator> [<param2>|"value"] with,
  - **param1, param2**: declared variables (local or from signature)
  - **operator**:
    - **a conditional operator**, <, >, ==, <=, >=,
    - **the operator "%%"**: which states that the rest of the division of param1 by param2 or value is zero.
  - **value**: a value coherent with the type of param1

3.6.4 Calling a function or sub-processing's

The parameters used in the parameter list must be either local variables of the caller processing or arguments of the processing.

In the following example, 2 functions (or processings) are called: "toto" and "tutu".

```
processing doALotOfThings(in Long my_param) begin
        parameter Long answer = "0"
        execute toto(my_param, answer)
        execute tutu(answer)
end
```

3.6.5 Control structures

The following example illustrates the control structures which can be used. First, an "if/then/else":

```
processing doALotOfThings_IFE(in Long my_param) begin
        parameter Long answer = "0"
        event e1 { my_param > "0" }
        if (e1) then
        begin
                execute toto(my_param, answer)
        end else begin
                execute tutu(answer)
        end
end
```

Then a "while":

```
processing doALotOfThings_W(in Long my_param)
        parameter Long answer = "0"
        event e1 { my_param > "0" }
        while (e1) do begin
                execute toto(my_param, answer)
                execute tutu(answer)
        end
end;
```

And finally, a "foreach":

```
processing doALotOfThings_W(in Long my_param, in LongList my_list)
        parameter Long answer
        foreach answer in my_list do begin
                execute toto(my_param, answer)
                execute tutu(answer)
        end
end
```

3.6.6 Calling a processing from an interaction

A processing is called from an interaction exactly as a user function.

3.7 Architecture definitions

Athéna provides a set of generic and basic elements, which have to be combined and specialised by the user to design a simulation. These elements allow the description of the structure, dynamics and flows.

The continuous domain components are:

- Parameter: a generic data container. Its data type is specified in the *Types' description View*;
- Interaction: used to compute the new value of parameters. They call a specific function or processing.

The discrete domain components are:

- State: represents the state of a complex object of the simulation;
- Event: event computed as a combination of states and conditions on parameters;
- Transition: allows a change of state. A transition is enabled when a given event is raised.

These elements are used to represent automata in the simulations.

Finally, for the structure of a simulation we have containers:

- Prototype: which can be mapped to classes in conventional Object Oriented Programming (OOP)
- Instances: of prototypes as you would have in OOP.

3.7.1 Containers

3.7.1.1 Prototype

A prototype is described as a container of other elements.

```
prototype <name> begin
    ...
end

prototype <name> is <ancestor> begin
    ...
end
```

Where:

- **name** is the name of the prototype. This name must be unique;
- **ancestor** is the name of an already defined prototype.
- **...** is the contents of the prototype. This can be anything except prototypes.

When a prototype inherits from another one, the consequence is the paste of the description of the ancestor at the beginning of its own description.

3.7.1.2 Instance

An instance of a prototype is described like this:

```
instance <prototype_name> <name>
instance <prototype_name>[] <name>
```

Where:

- **prototype_name** is the name of prototype;
- **name** is the name of the instance.

This name must be unique for the current context (prototype in which the instance is declared). This allows the modelling of an aggregation relation.

The '[]', if present, means the instance won't be unique, instead it will be a collection of instances referenced by a string identifier.

3.7.2 Continuous domain

3.7.2.1 Parameter

The general format for the description of a parameter is:

```
parameter <type> [<constraint>]? <name>
parameter <type> [<constraint>]? <name> [= "<value>"]?
parameter <type> [<constraint>]? <name> [= <>]?
```

Where:

- **Type**: is the type name of the parameter; the type can be completed with the following optional **constraints**:
  - '!', tells the generated simulator that multiple writers are allowed on this parameter

  - '#(*nb*)', tells the generated simulator that *nb* writers are allowed on this parameter

- **Name**: is the name of the parameter.
  - This name must be unique for the current namespace.
  - Two parameters can have the same name if they are included in different instances;
- **Value**: is the optional initialisation value of the parameter. If the type of the parameter is a user one, the initialisation value must be a string, parsed by a user function.
- **"= <>"**: is optional and means that the initial value will be given at initialisation by reading an XML file.

Example:

```
parameter Long colour
parameter Long level = "4"
parameter Double temperature = "-5.3"
parameter Boolean active = "1"
parameter String Name = "my Name"
parameter String toto = <>
parameter TrackList! tlist
parameter SensorStatusList #(3) tlist
```

3.7.2.2 Reference

A reference allows to use in a prototype instance a parameter defined in another prototype instance. The general format for the description of a reference is:

```
reference '<'type'>' [@]?<name>['.' <namei> ]*
```

Where:

- **Type**: is the type name of the parameter;
- **@**: if present, tells that the referenced parameter must be looked for by going up the instance tree.
- **Name**: is the name of the parameter or the first part of the relative path allowing to find the referenced parameter.
- **Namei**: A path is given as a dotted list of names, the last name being the name of the referenced parameter

3.7.2.3 Alias

This functionality has not been re-implemented in this version. An alias allows using a field in a structure type parameter:

```
alias ['<'type'>']? <name> : <structname> -> <fieldname>
```

Where:

- **Type**: is the type name of the parameter;
- **Name**: is the name which will be used for the aliased structure field.
- **Structname**: is the name of a structure type parameter or referenced parameter
- **Fieldname**: is the name of the alias field in the pointed structure.

This is syntactic sugar and avoids the user to manipulate structure when he only needs a specific field.

3.7.2.4 Interaction block

The description of an interactions looks like:

```
   [(when <stateset>::<state>)|(if <event>)]?
   begin
       interaction <function_name 1> (<param1>, .., <paramN>)
       …
       interaction <function_name N> (<param1>, .., <paramN>)
   end
```

where:

- **function_name 1...N** are the names of the function to execute at each activation of the interaction. These names must match the name of a declared function or processing;
- **param1, paramN** are the names of parameters or referenced parameters or alias structure fields used as arguments of the function. This list can be empty;
- **WHEN** <state> is optional. It defines activation conditions depending on state of an instance.
- **IF** <event> is optional. It defines an activation condition depending on an event.

The ‘period’ specificator has not been re-implemented in this version at the moment.

3.7.3 Discrete domain

3.7.3.1 Event

The format to define events is:

```
event <name> { cond1, …, condN }
```

Where:

- **Name:** name of the event. This name must be unique for the current namespace;
- **cond1..condN:** conditions. A conditition is: <param1> <operator> [<param2>|"value"] with,
  - **param1, param2**: declared parameter or referenced parameter or aliased struct field.
  - **operator**:
    - **a conditional operator**, <, >, ==, <=, >=,
    - **the operator "%%"**: which states that the rest of the division of param1 by param2 or value is zero.
  - **value**: a value coherent with the type of param1

Example:

```
parameter Double voltage 3.5
parameter Long reset = 0
event reboot {voltage == 1, reset == 1}
```

3.7.3.2 Signal

Signals can be emitted when transitions are crossed. The format to define a signal is:

```
signal <name>
```

Where:

- **Name:** name of the signal. This name must be unique for the current namespace;

See paragraph on transitions for a more complete example.

Example:

```
signal switchOn
```

3.7.3.3 State

The states are described as set of exclusive states:

```
stateset <stateset_name> { [<state>]+ } = <stateI> begin
         [<transition>]*
end
```

Where:

- **stateset_name**: is the name of the set of states. This name is only used to distinguish the different state sets and to help the user;
- **state**: are the names of the states. These states are exclusive and their names must be different;
- **stateI**: is the initialisation state. It must be one of the states of the list.
- **Transition:** liste des transitions

The initial state can be changed at top level after its definition (*not implemented in current version*):

```
<stateset_name> = <state>
```

The state and the stateset must already exist, and the state must be contained in the stateset.

Example:

```
stateset activation {off, on, down} = off begin
        …
end
activation = on;  // not implemented !
```

3.7.3.4 Transition

The transitions are described like this:

```
transition from <stateI> to <stateF> on <event> [raise <signal>]?
```

Where:

- **stateI**: is the name of the initial state. This state must have already been defined;
- **stateF**: is the name of the final state. This state must have already been defined. The initial and the final state must be contained in the same stateset;
- **event**: is the name of the event which triggers the transition;
- **signal**: is the name of a signal which is raised upon the transition

Example:

```
signal switchOn
signal switchOff
```

```
stateset button { b_off, b_on } = b_off
begin
      transition from b_off to b_on on switchedOn raise switchOn
      transition from b_on to b_off on switchedOff raise switchOff
end
```

3.8 Simulation specific definitions

3.8.1 Library mapping

3.8.2 Compilation specific data

4. GENERATION

At the moment, the tooling around the ADL consists into two generators:

- A C++/Matlab simulation and XML configuration generator
- A C++ wrapper generator for models

In the following paragraphs we present these tools.

4.1 Principles of the generators

The user ADL files are first parsed. This parsing phase allows the construction of a tree containing the following data:

- Header information
- Types' declarations
- Functions' declarations
- Processings' declarations
- Prototypes' declaration
    - Parameter, references, aliases, state sets, events,
    - Interactions declarations,
    - Instance declarations.
- Defines and Mapping information.

From these data, the generators apply different templates in order to generate a simulation or model wrapper. The "CodeWorker" tool is used for these 2 phases.

4.2 Simulation generation

The tools presented in this chapter are at the moment:

- In use on Windows XP 32b with the Visual Studio .NET 7 compiler.
- Compatible with Linux and the gcc 4.0 compiler (re-validation needed for this platform).

4.2.1 Structure of a generated simulation

A simulation consists in a set of initialisations, a main loop, the generated source and user functions:

1. Look into program options: the configuration filename can be chosen and the number of simulation steps also.
2. Load configuration file.
3. Load user libraries: from configuration file, load all specified user libraries. Each library declare the user functions it services.
4. Instantiate the instances of a simulation
5. Initialize the connections between the instances (referenced parameters)
6. Initialize the instances from the configuration file
7. Execute simulation: loop for the specified number of steps.
8. Delete simulation

4.2.2 Generated Types

An include file named “TypeDefs.h” is generated from the ADL specification. For basic types, an example:

```
basic Long
        LANGUAGE <CPP> : “long” = “0”;
END;
array<Long> LongArray [10];
vector<Long> LongVector;
```

```
map<Long, Long> LongMap;
```

Is translated to:

```
// Long

typedef long Long;

XmlNode* Long_toXML(const Long& l);

Long Long_fromXML(XmlNode* xn);

Void init_Long();


// LongArray

typdef long LongArray[10];

XmlNode* LongArray_toXML(const LongArray la);

Void LongArray_fromXML(XmlNode* xn, LongArray la);

Void init_LongArray();


// LongVector

typedef std::vector<Long> LongVector;

XmlNode* LongVector _toXML(const LongArray la);

Void LongVector _fromXML(XmlNode* xn, LongArray la);

Void initLongVector();


// LongMap

typedef std::map<Long, Long> LongMap;

XmlNode* LongMap _toXML(const LongArray la);

Void LongMap _fromXML(XmlNode* xn, LongArray la);

Void init_LongMap();
```

As can be seen, the translation consists in:

- A typedef
- Functions to serialize/deserialize to/from XML.
- An initialization function which takes into account the user init values.

If Matlab was used, functions to translate from/to Matlab would also be present.

For structured types, a C/C++ class is created, for example:

```
struct FlightParameters
        Double lat;
        Double lon;
        Double alt;
end;
```

Is translated to:

```
class FlightParameters {
public:
        Double lat;
        Double lon;
        Double alt;
        FlightParameters() {}
        FlightParameters(const& FlightParameters _fp) { copy(_fp); }
        Operator = (const& FlightParameters _fp) { copy(_fp); }
};
XmlNode* FlightParameters_ toXML(const FlightParameters& fp);
Void FlightParameters_fromXML(XmlNode* xn, const FlightParameters& _fp);
```

The generated class contains:

- The fields,
- The empty and copy constructor
- The operator =

As for simple types, (de)serialization functions are provided.

The generated file is included in all generated definition files.

For dynamic arrays and structure, an allocate method is also added. This method is called at simulation initialization with the user inputs.

4.2.3 Generated Instances and prototypes

A prototype maps to a C++ class which contains all the elements present in the prototype. If the prototype inherits from a parent, then the parent characteristics are also introduced in the C++ class.

As a consequence, an INSTANCE maps to an instance of a C++ class. If a collection of INSTANCE is specified, then it maps to a collection of instances of a C++ class.

Each generated C++ class inherits from a simulation component class. This class provides the needed services for incorporating an instance in a generated simulation:

- Set/Get parent in instance tree
- Add/get child instances
- Initialization
- Connection of references
- Simulation of a Step

Otherwise, the generated class includes:

- accessors on its parameters, referenced parameters, aliased struct field, events, state sets.
- interaction methods for each declared interaction.

4.2.4 Generated Processing's

A processing is represented by a C++ void function. The arguments of the function are defined as:

- “const <type> &” : if the argument has been declared “IN”

- "<type> &" : if the argument has been declared "INOUT" or "OUT"

The local parameters are first declared. The other processing or user models are then called.

For example,

```
Processing  tutu (in Long i, out Long j)
        execute user (i, j);
end;

processing toto (in Long i, inout Long j)
        parameter Long k = "0";
        execute tutu (in Long i, out Long k);
end;
```

Is:

```
// Processing tutu
void tutu(const Long& i, Long& j) {
        user(i, j);
}

// Processing toto
void toto(const Long& i, Long& j) {
        Long k = 0 ;
        tutu(i, k);
}
```

4.2.5 Parameters

A parameter of type <T> is defined as an instance of a template called "Parameter". This template has the following methods:

- *"const <T>& getRead(int date)"* : return the read parameter (written before current date).

- *"<T>& getWrite(int date)"* : return the write parameter (needed for optimisation purpose).

- *"void write(int date, <T> value)"* : write the current date value of the parameter.
- *[0] or [1]: allows access to the 2 buffers (Read/Write) for initialization purpose.*

4.2.6 References

A reference is a pointer to a parameter. This pointer is resolved and set at simulation "connection" phase.

4.2.7 Event

An event is the composition of conditions on basic or calculated parameters. It is represented by a Boolean function. For example,

```
event initializing { time == "0" };
```

Is:

```
bool instance_name::event_initializing(const long& icycle) {
        bool result = true;
        result = result && (get_time(icycle) == 0);
        return result;
}
```

4.2.8 State Set

A state set is represented by a parameter template on an "enum".

```
stateset ModelState {INIT, RUN} = INIT;
```

Is :

```
enum stateset_ModelState { INIT, RUN };
Parameter<stateset_ModelState> ModelState ;
```

When used :

```
if (ModelState.read(icycle) == INIT) {
}
```

4.2.9 Interactions

An interaction is represented by a void function which encapsulates the call to the user function or processing. WHEN and IF conditions are added if present before the model or processing call.

For example,

```
interaction model:
  when (RUN),
                    callSensorModel (time, fp, so , im);
```

Is:

```
void instance_name::interaction_model(const long& icycle) {
 if (ModelState.read(icycle) == RUN)
   {
        Long _time = get_time(icycle);
        FlightParameters _fp =get_fp(icycle);
        SensorOrders _so = get_so(icycle);
        Image _im =get_im(icycle);
        callSensorModel(_time, _fp, _so, _im);
        set_im(icycle,_im);
   }
}
```

Arguments are copied if modifier is “INOUT”. They must have a "Deep Copy" constructor ! This particular constructor is generated for types described with the ADL (“=” operator and copy constructor).

4.2.10 User Model Stubs

4.2.10.1 Generated stub

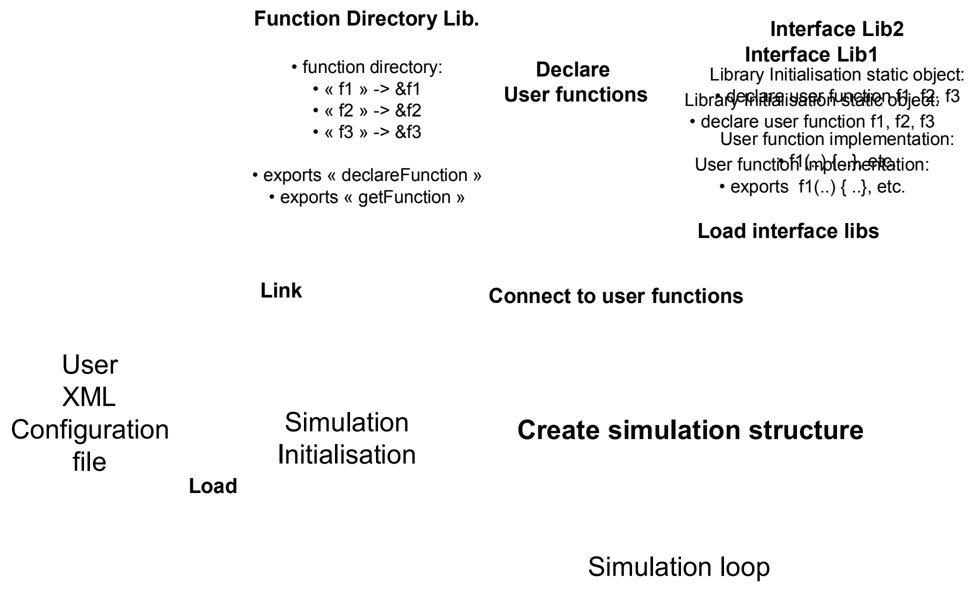
For each programming language used in the simulation, an Interface library is generated The Mapping definition file allows the generation of the dependencies. An example is given here after.

```
Libraries <CPP>

        LINK "../simutools/libSimuTools.lib";

        LINK "../flightAndLoc/libFlightAndLoc.lib";

        LINK "../sensors/libBasicSensors.lib";

        LINK "../communications/libBasicCommunications.lib";

        LINK "../mms/libBasicMMS.lib";

        LINK "../threats/libThreats.lib";

END;
```

4.2.10.2 User dynamic libraries loading

The following schema illustrates the dynamic loading of user libraries:

- Step 0 consists in the link with the library containing "user function directory services". This directory is a static map of function pointers referenced by their name. Two services are available, declare a function and get a function.
- Step 1/2, the simulation loads the xml config file.
- Step 3/4, it loads each interface libraries as specified in the config file.
  - In each library, a static object called "StubLibNameDLL" is instanciated at the library initialisation. This object constructor calls the "declare function" service for each user function implemented by this library.
- Step 5/5', the simulation components are instantiated, connected and initialized. At initialization , in each component, user interactions are mapped to user functions by calling the function directory.


Function Directory Lib.
• function directory:
• « f1 » -> &f1
• « f2 » -> &f2
• « f3 » -> &f3
• exports « declareFunction »
• exports « getFunction »
Declare
User functions
Interface Lib2
Interface Lib1
Library Initialisation static object:
• declare user function f1, f2, f3
Library Initialisation static object:
• declare user function f1, f2, f3
User function implementation:
• f1(..) { ..}, etc.
User function implementation:
• exports f1(..) { ..}, etc.
Load interface libs
Link
Connect to user functions
User
XML
Configuration
file
Load
Simulation
Initialisation
Create simulation structure
Simulation loop

This mechanism is portable on Win32 VC7 / Cygwin GCC 3.4 / Linux RH9 GCC3.4.

The "isolation" of the function directory services in a particular library is needed for the portability to windows.

This mechanism implies that each user library instantiates a static object or implements a "DLLMain" in which it declares the user function it implements.

4.2.10.3 C/C++ User models

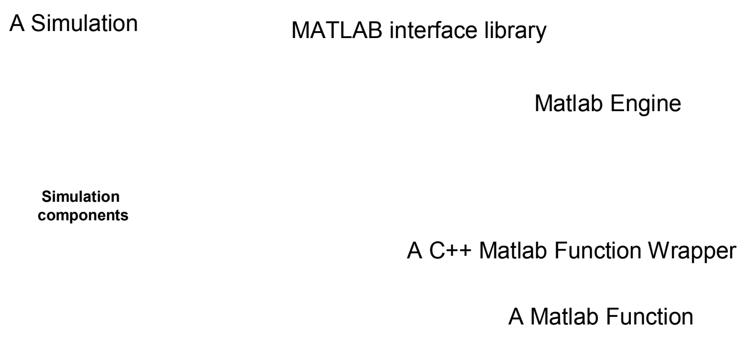
C/C++ User models must implement the interface of their models as defined in the ADL. The API-Model generator [4.3] allows the generation of these interfaces.

4.2.11 MATLAB User models

The Matlab interface allows:

- To map C/C++ data to Matlab data and vice-versa: the data types must abide to the following rules:
    - Simple types: string, double, int, Boolean,
    - Arrays of simple types with at max 2 dimensions.
    - Structured types containing:
        - simple types,
        - Arrays of simple types with at max 2 dimensions,
        - Structure types.
- To call Matlab functions from C/C++: the Matlab function signature will have to be the one described in the ADL.

The following figure illustrates the way Matlab functions are encapsulated.



4.2.12 Generated configuration files

The generated configuration file structure is an image of the instance tree of the simulation. For each instance, each configurable parameter is listed. It also contains the list of user libraries which must be loaded during initialization.

4.2.13 Execution

To run a simulation, create an XML configuration file from the generated template ("configTemplate.xml") then execute:

Simu.exe –config <configFileName.xml> --nbSteps <Simulation nbr of steps>

4.3 API Model Generator

The API Model allows the specification of the interface of a model in order for this model to be compatible with Athena. The API Model generator allows the generation of the interface source code from the specification.

4.3.1 Specification

This specification principles are equivalent to the one presented for the simulation. A model specification consists in the following items:

1. A header describing the model and referencing other parts of the specification
2. A set of data type specification as described in [Error: Reference source not found]
3. A set of function signature specification as described in [Error: Reference source not found]
4. A set of processing description as described in [Error: Reference source not found]
5. A set of defines
6. A set of objects which are part of the model

Items 2, 3, 4 and 5 are the same as the one used by the simulation.

4.3.2 Generator

The generator generates:

- o The interface skeleton with all model entry points
- o The object file skeleton with their header.
- o The Makefile to compile the library