Gerd Doeben-Henisch, member PES

The Planet Earth Simulator Project – FSL Vers.1.0.1

(Last modified: May-18, 2004, 13:00h)

***Abstract*—This documents describes the FORMAL SYSTEMS LANGUAGE (FSL) of the PLANET EARTH SIMULATOR (PES) Project Version 1.0.1. It is describes the 'use case' of the language as well as the grammar and some examples.**

***Index Terms*— modelling real world, simulator, world, cognitive agents, automated parallel computing, safety critical**

INTRODUCTION

The FSLLanguage of the PESProject is a language for *modelling real world processes* and *cognitive agents*.

The FSL is also constructed with *real time processing* and *automated parallel processing* in mind. It also provides basic mechanisms for the construction of *safety critical systems*!

The main scenario in mind for the application of the FSLLanguage can be seen in the following figure.

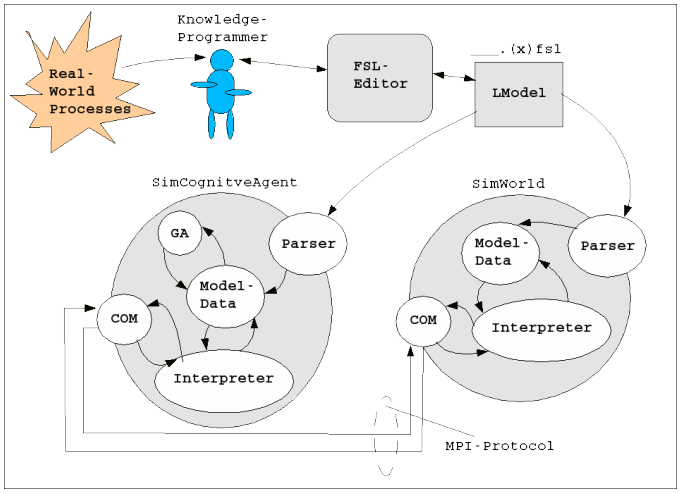

Fig. 1: Main Application Scenario for FSL-Language

A *knowledge programmer* maps some real world processes into a logical model, a so called *LModel*. The language of the LModel is either *'pure'FSL (suffix .fsl)* or an *XMLversion (suffix .xfsl)* of the FSLLanguage.

The aid to generate an LModel is some kind of an *editor*. This can be an *ordinary text editor* or some dedicated editor, e.g. FSL-TEdit or FSL-VEdit. The *FSL-TEdit* (see [Robert VIRGA 2004]) is a dedicated text editor which has not only the text of some LModel 'buffered' but it works with the complete definition of the language in the background, and with a detailled data structure reflecting all the syntactic properties of the language. Thus this editor is able to support the knowledge programmer in a deep way to construct wellformed LModels. In the future allows this kind of a syntax driven editor very useful functions (like e.g. the famous GNU-editor emacs).

FSL-VEdit is a visual editor project which should allow in the future the construction of an LModel by editing of graphical symbols (cf. [Gerd DOEBEN-HENISCH 2004b]).

The XMLversion of FSL will be generated by converting an .fsl-file into a .xfsl-file. There will be at least one converter available *FSL2XML* which will be bidirectional (see [Volker LERCH 2004]). The FSLVEdit as well as the FSLTEdit will work with .xfslFiles.

The main 'users' of LModels are socalled *simulators*. At the time of this writing there are two main types of simulators: *SimWorld* simulators and *SimCognitiveAgent* simulators.

A *SimWorld* simulator takes an LModel as input, *parses* this file and generates a SimWorld *Data Model*. This Data Model will then be processed by the SimWorld *Interpreter*. The interpreter is also communicating with outside clients through the *MPI-Protocol* (for the MPI-Protocol see [William GROPP 1999] and [Tom GEHRSITZ 2004]).

A *SimCognitiveAgent* simulator is structurally similar to a SimWorld simulator, but there is one component, by which it differs. The SimCognitiveAgent simulator has additionally a component called *GA* representing a *strong learning component* (see [Gerd DOEBEN-HENISCH 2004a]. Usually GA is realized by some genetic Algorithms but this must not be so; it can be any other mechanism. The strong learning component takes the Model Data as input and changes these 'slightly'. The output is therefore a modified Data Model.

The important point here is that the Data Model in case of the SimCognitiveAgent simulator is not the logical model of

The PES-Project is supported by the Institute for New Media e.V. in Frankfurt am Main (Germany)

any world process but of those world *processes* which describe the *structure of cognitive agents* as seen by *cognitive psychology and neuro psychology*. Thus these Data are complex dynamical structures representing a weak learning function (cf. [DOEBENHENISCH 2004a]). It is this twofold scenario World and CognitiveAgentwhich gives the special 'flavour' for the design of the FSL.

THE FSL SURFACE

With the FSL 'SURFACE' we mean the main concepts which are visible to the user of the FSLlanguage. As in the object oriented paradigme, where you have classes as the main concepts around which all the other concepts are organized, you will have in the FSLlanguage *systems* as the main concept.

A system is a structure with a set of inputs *I*, a set of outputs *O*, some internal resources *R* and a system function *F*.

SYS(x) iff x = <<I,O,R>, F>
with
F: I x R > R x O

The *epistemological assumption* behind the FSLlanguage is therefore, that you can map all processes of the real world either into one single system or into a system which recursively contains other systems as elements.

A system can be interpreted in many different ways. Two special interpretations will be mentioned here because these are of special importance in the case of the modelling of cognitive agents: neuron and rule.

Every system can be interpreted as a *neuron* or a network of neurons. And because there is no constraint to a certain kind of neuron you can built every kind of neuron wih this formalism.

Also can a system be interpreted as a *rule* where the system inputs are the conditional variables and the system outputs are the resulting actions.

Clearly is it also possible to map the concept of *fuzzy concepts* into the system concept.

Thus the decision to use the FSLlanguage for processing does not necessarily imply a limitation in expressive power.

MORE ABOUT USING THE FSL

To use the system concept as central term in the FSL-language induces some need to decide how to write it. We will use he following convention:

system sys_name (input parameters ...) [output parameters ...] sys_body

The *input parameters* are describing the input of the system and the *output parameters* the output. From the outside nothing else is known of a system than these values! This is the whole interface. This is different to the class concept in object oriented modelling. A *class* can contain methods which can be called from the outside. This is not possible with systems. With systems you can only deliver values for input or you can receive values from output (for the point of communication between systems see below).

At one side this is a drawback because you can not 'use' the system from the outside, but on the other side it is a 'pleasing' because you can capsulate complex machineries which in turn allows you to *'cut' comlex systems easily into subsystems* without any further operations; you only have to support the exchange of input and output values of the divided subsystems.

Because it is planned to provide *parallel processing* of the simulators in an '*automated fashion*' it is especially this feature of simple 'cutting' complex systems into subsystems which is very promising for the future applications.

Input parameters as well as *output parameters* consist of a finite sequence of *parameters*. A *parameter* is understood as a triple concept:

type par_name unit

A type *is a basic data type* or a *defined data type* (like 'int', 'float' 'bool' or 'string').A *par_name* is an identifier. The *unit* is a unit like 'year', 'month' ,'day' etc describing some measurement unit primarily for time; but it can be extended to other units of measurement as well.

A special situation arises if there is more than one output parameter. In this case must the 'value receiver' be of a format which allows the reception of such values. If e.g. there is a sequence of output parameters like [int a, float b] then the receiver must also be a list with two parameters of the same type. From this follows that the existence of finite sequences of output parameters presuposes the availability of a *struct.* A struct is a list with a fixed set of typed elements.

struct name { type1 name1; ... typek namek; };

Another variant would be a *typed array of fixed length*:

type name[dim];

From this we conclude that a statement like

x = function(p1, ...)[q1, ..., qk];

can only be true if the receiver 'x' has a type which is conformant with the finite list of output parameters, and this can only be a *struct* because only in a struct there is a definite order of the elements and every element is typed like the individual output parameter. Thus we presuppose

struct x { type1 q1; ... typek qk; };

Clearly one could realize this also with a *list*, but the concept of a list is much more ' liberal' which induces more computational load into the actual processing. Thus we will stay with the more definite concept of a *struct* especially also because the input and output of function should be as fixed as possible with regard to *substitutability*.

The FSL-language suppports also basic mechanisms for the construction of *reliability* and *safety* informations. This is done by claiming that every system has to have an automatic *hidden* communication of the *error status* of the delivering system. The error state is of type *struct* and has to be defined as a *built in structure.* The error state has to be designed in a way which tells the observer *who when which error* has to communicate. The error state is hided from the user but is automatically processed by the *interpreter*. In case of errors will the interpreter write dedicated error-messages to the console and into a logfile. With this mechanismen a direct and explicit error handling is possible.

The next point is the distinction between declaration --or definition-- and calling --or usage--.

Befor some entity can be used within an FSL-system it must be *declared (defined).* If an element is declared it can be *used (called). Built-In* elements are declared by default. Thus, they can be used from the beginning.

Which types of elements do exist in the FSL-language? In version 1.0.1 there are only *systemcalls* and within systems can exist besides other systemcalls only *states*. States are either *simple* like

int aA;
float bB;

or they are *compound/ complex* like

struct aaAA { int X; float Y; struct bbBB { ... }; .... };

struct aaAA name[100];

If there is a *systemcall* of a system s1 which is not based on a builtin system then there must exist somewhere outside the system which contains the call of system s1 a *systemdeclaration* of system s1. Thus

sysgen (a)[b]{ ....
sysspecial(str)[bbb]; ... }

system sysspecial (string str)[string bbb]{ .... }

Systems occuring as systemcalls within other systems are called *subsystems* or *embedded systems*. Embedded systems receive their input from the *surrounding system* and they deliver their outputs to the surrounding system.

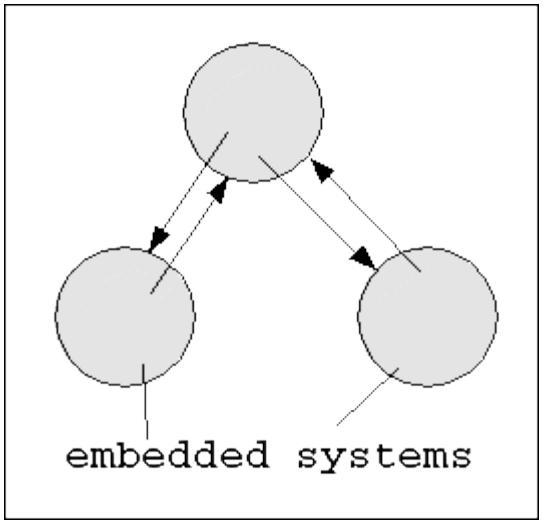

Fig. 2: Embedded systems in surrouding system

The set of systemcalls can further be divided into *ordinary* system calls and into *meta* system calls.

Meta system calls like

if(condition)[]{...}
if(condition)[]{...} else { .... }
while(condition)[]{ ... }

are formally system calls but they function like a *switch.* If the *condition* has the boolean value *true* then the system calls in the curley brackets will be executed; otherwise, if the value of the *condition* is *false* then the system calls in the following brackets will not be activated. Thus in the case of an *ordinary* system call there is always an execution not so in the case of the *meta system calls*.

Conditions can be expressions which can be computed with regard to a *boolean value*. This is only possible if one uses either states with a *boolean* value or system calls with a *single boolean value as output parameter.* Examples of such *boolean system calls* are:

less(x,y)[]; --- also written (x < y) ---
equal(x,y)[]; --- also written (x == y) ---
greater(x,y)[]; --- also written (x > y) ---
lequ(x,y)[]; --- also written (x <= y) ---
gequ(x,y)[]; --- also written (x > =y) ---
etc.

INPUT-OUTPUT COMMUNICATION

An important and not quit simple task is the inclusion of some input-output operations.

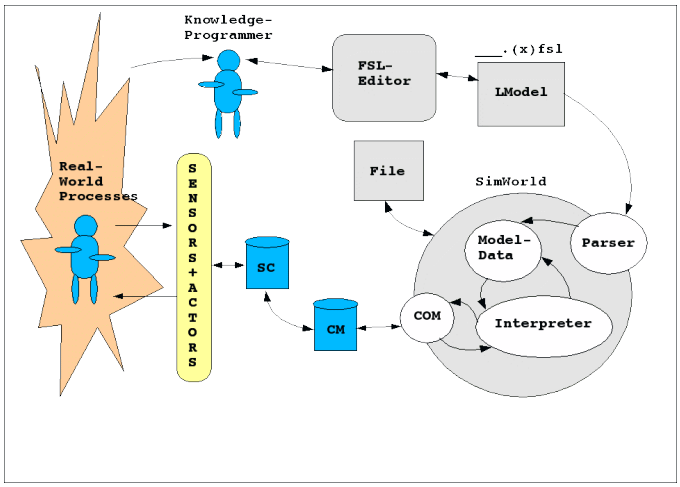

Fig. 3: Use case for input-output communication

To understand the problem one has to consider the applicatoon scenario for input-output in the case of the PES-simulators.

The usual case will be that a human user starts a simulator for a certain topic. In most cases the LModel for that topic is parametrized, i.e. before starting the processing of the LModel process the LModel will ask for some values which will by used as *parameters* to shape the intended process in the light of the user' sintention.

As discussed in the PES-team during April 2004 it should be the duty of the PES *Simulation Control (SC)* Server to interact with the user in a way that after the selection of an appropriate LModel for simulation the SC will automatically ask the user for all initial values needed for the start of the LModel.

The result of these negotiations with the user is an *simulation contract (SimCon)* which contains all these informations and wich will be delivered to the simulator immediately after starting the process.

This assumption will work if one presupposes that the *interpreter of the simulator* takes the simCon and delivers all needed values to the program. In this case one could assume that the startup parameters are coded as *input parameters of the root-system.*

A quite different situation is given when during the simulation process some *information is needed from the outside of the system* or some *sensors from the outside have actual information which has to be transmitted to the system.* How can this be handled?

Within the PES-Project there is only one answer: *by communication*! The interchange of information between system will be handled by communication which is realized by *sending* and *receiving messages*!

As mentioned before will the PES-Project in the first phase use the MPI-Protocoll as framework for these communications between systems, at least between the different types of simulators. The *Cluster Manager (CM)* functions here as a kind of Master and as a *gateway* of the MPI-Protocoll Area into other communnication protocols (see [Tom GEHRSITZ 2004]).

Thus the only mechanisms which are needed within an LModel are at least two functions:

send()[];
receive()[];

The question is what kind of parameters these functions do need. For version 1.0.1 we assume the following format of the string, which is the only input parameter of the *send()[]- and receive()[]-call:*

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- $Id$ -->
<Command>
  <Source>0</Source>
  <Destination>1</Destination>
  <Type>ActionString</Type>
  <Parameter>ValueString</Parameter>
  <Timestamp>2004/05/16 10:00:03</TimeStamp>
</Command>
```

The system call for the receive action presuppoeses that there exists a dedicated *managed queue for incoming messages* which can be read ' atwill' all the time.

Wheras the *sending-action* can actively be started by a system is the *receiving-action* somehow 'passively' ; the receiver can only *wait* and then, when some *message arrives* can the receiving system read the received message. What is needed here is some kind of an *event system* which triggers the waiting system.

Seen from the perspective of *real time* systems which have to be *reliabel* and *safety* it could be necessary to force the system into a deterministic scheduling of evaluating some message queue *regularily*! This would imply the installation of a *deterministic loop* with a *transparent timing behaviour.*

In the case of *Safety Critical Systems* (including Real Time Systems) it is necessary to include the mechanisms of the used communication protocol into the design of the whole system -- not to forget the interpreter, which is doing the main job of processing--.

In Version 1.0.1 of the FSL we will assume a simple

schedukling-based version of the usage of *send()[]* and *receive()[]*. This means that the programmer of an LModel can place *send()[]* and *eceive()[]* like ordinary built-in system calls everywhere in his LModel.

MEMORY MANAGEMENT

As in nearly all programming languages there exists also in the FSL-language some need for a *dynamic memory management*. At some time during the processing of an LModel it can happen that an event claims for some memory to store data.

Usually the memory management system call needs to know what *type of data* has to be stored and *how many items* should be reserved. And then it will return some kind of an adress or pointer to this new item.

Seen from the usage of the language it would be most comfortable for the user that he/she had not to worry about the details of memory management. The most simple behaviour would be that within the LModel at any time it would be possible either to *introduce* a new state or to *change* an already existing state.

Thus like in the case of an array it should be possible either to declare a certain dimension in the beginning

int array1[5];

or to let it unspecified

int array2[];

and then during the process use the array as necessary, i.e. if you *add* an item which is not already there it will *automatically be added* and if the item is already there then it will be *overwritten*.

This *automatic handling* of memory handling would deliver the control of the correct usage of the memory to the interpreter. This would exclude possible memory errors like in languages as C and C++. If there would not enough memory be available then the system should not execute the operation and generate an *memory exception event*!

This automatic handling of the memory should apply also to *strings* but probably not to *structs*.

The only way to hinder a dynamic modification of a state should be to use an additional keyword *const* to signal that this state should always keep his values fixed.

Adding of items then could be managed as follows:

Given some array and some string:

int array1[5];
float array2[];
str L1 = "Hallo World!";
str L2;

one could add items as follows:

array1[6] = 200;

This *adds a new item* at position 6 of the array, i.e. the array must automatically be extended.

array2[3] = 7.345;

This *overrides* the item at position 3 with a new item.

array1[2] = 20;

This *adds a new item* because the original string had no item.

concat(L1," This is an extension.")[];

This extends the old string by *concatenating* a new sring at the end of the old one.

subst(L2,3,5,"XXXX")[];

This replaces the symbols at positions 3-5 in the old string by a new string "XXXX" by the subst()-call. The result should look like this:

"HaXXXX World!"

L2 = "Any string can be replaced.";

Direct assertion *replaces* the old value *completely* by a new value.

SYSTEM ARGUMENTS

BY VALUE OR BY REFERENCE?

As known from languages like C and C++ it is possible to pass a *reference* to a state instead of making a copy of the whole data structure.

Calling by reference has some advantage, at least you will need less memory space and it is usually faster.

If more than one process can operate on the same data structure --which can happen in the case of FSL with parallel processing-- then a call by reference from more than one subsystem can lead to *conflicting operations.*

If in the future indeed certain subtasks can be *cutted* from the parent tasks for processing on concurren processes then it could be an advantage that every subprocess has its *own copy* of the datastructure. Then every subprocess could operate independently on the other concurrent processes.

But also in this 'copy-case' there could arise a conflict if different concurrent subsystems would operate on the same piece of data structure. It would e necessary to have some kind of *merging procedure.*

If one would work with calling *by reference* and one would try to solve concurrent processing by *sending messages* then would this kind of procedure mimic the situation of having only *one single* process thus giving up the advantage of having distributed concurrent processes running.

Therefore, as long as one can assume *one single processing unit* it is more advantegeous to use *call by reference.* If there are concurrent processes it could be of help to have *call by value* with the need of *monitoring possible differences* between shared portions of the data structure.

Otherwise, if one votes for *calling be reference* and one would allow the distribution of processes then it would be anyhow necessary that the difeerent distributed processes will *communicate by messages.* A communication of the values of a referenced data structure by messages would then not induce a quite new mechanism. There would be remain the need for a monitoring of concurrent operations.

Thus for FSL 1.0.1 we will assume the possibility to use both: call by value and call by reference.

sysany(&A,B)[];

The system call with sysany()[] calls the variable A by reference and the variable B by value.

EXAMPLE NR.1

Lets have a look to an example of an LModel which describes a simple SimWorld simulator simulating a simple world for possible inhabitants.

System sys1 establishes some datastructures and then starts a loop within which it calls repeatedly the queue of incoming messages with the receive() call.

Depending from the commands which will be found in the incoming string it will process these different commands. In the example there are only four commands 'login', 'logout', 'move' and 'eat' which will each be processed by a special system 'loginsys()', 'logoutsys()' etc. At the end of each processing is always a send()-call to inform the client about the reaction of the system.

What this figure also shows is the layout of the *implicit tree-structure of a LModel.* There is only one root-system with input and output. This root-system can recursively have other sub-systems. Processing is from the root to the 'children' and back from the children to the root. Between the different systems only values are communicated.

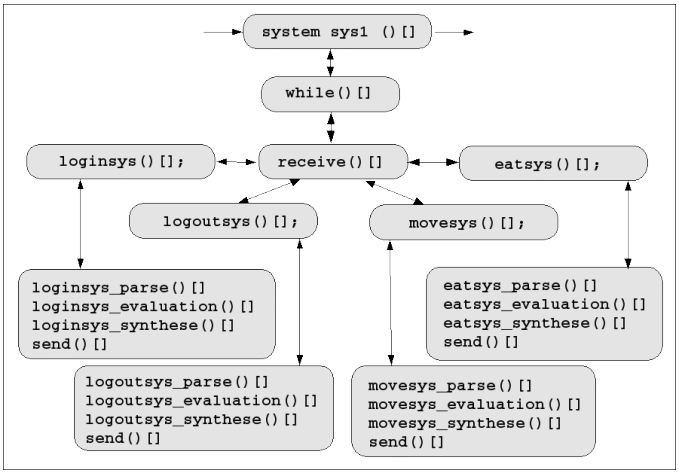

Fig. : Simultor simulating a simple world by system sys1

REFERENCES

(not finished; preliminary)

Francis COTTET/ Joelle DELACROIX/ Claude KAISER/ Zoubir MAMMERI [2002], "Scheduling in Real-Time Systems", John Wiley & Sons, Chichester (Engl.) et al.

Gerd DOEBEN-HENISCH [2004a], "Simulation of Knowledge", Seminar conducted during the summer term 2004, University of Applied Sciences Frankfurt am Main (Germany), URL: http://www.fbmnd.fh-frankfurt.de/~doeben/IS/SIMULATION-WISSEN/is-simwissen-header.html

Gerd DOEBEN-HENISCH [2004b], "The Planet Earth Simulator Project – First Considerations to the Visual Programming Interface of the Formal Systems Language", Draft Paper, Planet Earth Simulator Project

Tom GEHRSITZ [2004], Clustermanager based on the MPI-protocol, Dipl.Thesis

William GROPP/ Ewing LUSK/ Anthony SKJELLUM [1999 2nd ed.], "Using MPI. Portable Parallel Programming with the Message-Passing Interface", Cambridge-London: MIT Press

Volker LERCH [2004], FSL2XML (Dipl.Thesis)

Neil Storey [1996], "Safety-Critical Computer Systems", London - New York et al:Prentice Hall (Pearson Education)

Robert VIRGA [2004], FSL-TEdit (Dipl.Thesis)