Towards a Distributed Simulation Toolbox for Scilab

Awad Mukbil[1], Peter Stroganov[1], Umut Durak[1,2], Sven Hartmann[1]
[1] Clausthal University of Technology
Department of Informatics
[2] DLR Institute of Flight Systems
{awad.mukbil, peter.stroganov, sven.hartmann}@tu-clausthal.de, umut.durak@dlr.de

Scilab is an open-source cross-platform computing environment for engineering and scientific applications. It provides a high-level programming language with hundreds of built-in functions for numerical computation. Additionally, it encompasses Xcos, which is a Causal Block Diagrams-based graphical editor for modeling and simulation of dynamical systems. Thus, it supports model-based simulation. With emerging technologies like cyber-physical systems, ubiquitous computing, smart devices and ambient intelligence, the subject reality (simuland) involves multiple heterogeneous and distributed interacting entities. Hence, the modeling and simulation of these emerging systems is evolving towards constructing distributed simulations where individual models of distributed entities interact with each other via well-defined and agreed interfaces. Although distributed simulation has been widely utilized by mainly defense modeling and simulation community since the 1980s, the combination of distributed simulation with model-based simulation techniques for simulating technical systems poses new research challenges. To date, Scilab and Xcos do not contain built-in distributed simulation capabilities. This paper first introduces the requirement of distributed model-based simulation and then presents an implementation strategy aligned with the Scilab and the Xcos software architecture.

1 Introduction

Scilab is an open source, cross-platform computing environment [1]. It is widely employed for engineering and scientific applications such as signal processing or optimization [2].

Scilab encompasses a high-level programming language with hundreds of built-in functions for numerical computation and a graphical editor, called Xcos, for modeling and simulation of dynamical systems using Causal Block Diagrams.

Technical system is the term that is used for all man-made machines [3]. As the emerging category of technical systems, Cyber Physical Systems (CPS) are now addressing new capabilities and properties. CPS are real-time systems which are composed of distributed networked heterogeneous physical devices and computational components [4]. They were introduced as the integration of computing with the physical processes [5]. They are characterized by their networked interacting components.

Not only the technical systems, but their operation environment is also changing. Further ubiquitous computing concepts are integrating mobile computing capabilities with pervasive elements of the environment, including sensors, actuators and computing nodes [6]. With ambient intelligence, the environment is getting smarter and reactive [7]. Simuland is defined as the real-world item of interest; the object, process, or phenomenon to be simulated [8]. The simuland in the modeling and simulation of technical systems is evolving rapidly towards involving multiple heterogeneous, and distributed interacting entities. Hence, distributed simulation tools and techniques are more required now for co-simulating loosely coupled individual models of distributed entities of emerging technical systems. These techniques and tools are required in model-based simulation environments, like Scilab, that are extensively used in the engineering of technical systems.

Distributed simulation is a technology that enables a simulation to be executed on multiple computing nodes, such as a set of networked personal computers [9]. It usually deals with the execution of simulations on loosely coupled systems. It includes execution on geographically distributed computers interconnected via a wide area network such as the Internet. While the traditional motivations were reducing execution time, enabling geographically distributed experimentation, integrating simulators from different manufacturers and fault tolerance, recently, the distributed nature of simulands and the corresponding models

has been declared the fundamental purpose of distributed simulation [10].

The communication among the entities in a distributed simulation can be conducted through pair-wise connections or by utilizing a shared bus (middleware) that all simulations can use collectively. High Level Architecture (HLA) proposes industry-wide accepted standardization in a shared bus by specifying the interfaces with the interconnected entities and the shared bus [11][12][13]. There are also some efforts that focus on providing HLA functionalities in model-based simulation environments. Pawletta et al. propose a HLA toolbox for MATLAB [14][15]. ForwardSim is providing HLA Toolbox for MATLAB [16] and HLA Blockset Simulink [17], while Theppaya et al. offer an approach to integrate HLA Runtime Infrastructure Services with Scilab [18].

These efforts constitute bases for constructing an HLA capability in a model-based simulation environment. While it is a legitimate requirement to support HLA for a distributed simulation toolbox, due to high complexity and a relatively gradual learning curve, simple and relatively easy pair-wise networking is regarded as the initial capability set towards a more capable distributed simulation toolbox. Therefore, this study presents an attempt at enabling pair-wise connections between simulation entities, particularly in Scilab and Xcos.

One of the widely employed techniques for pair-wise networking simulations is to use Transmission Control Protocol / Internet Protocol (TCP/IP) or User Datagram Protocol (UDP). Accordingly, this paper will present an implementation strategy for UDP and TCPI/IP networking features in Scilab and Xcos aligned with their software architecture. Section 2 presents an implementation strategy for a scripting level networking Application Programming Interface (API) that is designed as an external Scilab module. Then, Section 3 introduces an approach for developing a networking block library as an Xcos palette.

2 Scilab Networking Module

Scilab supports several general purpose programming languages, such as C/C++, Java or FORTRAN. The Scilab Networking Module is proposed as an external Scilab module. It aims to constitute an abstraction layer for Scilab users. This abstraction layer will wrap BSD Sockets [19], a classical socket API that was developed in the 1980s.

Briefly, the Scilab Networking Module will give Scilab the ability to open a socket by choosing/reserving a Port number and the local IP address of the device. Then, based upon the protocol that will be used (TCP/UDP), Scilab will start communication in a Client/Server architecture.

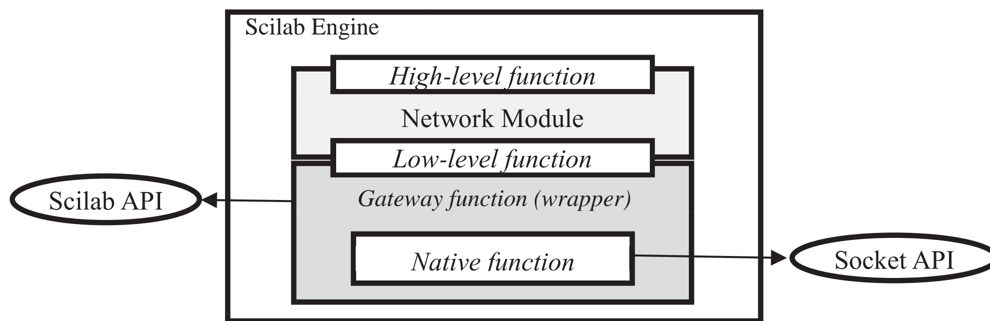

Figure 1 Extending native function to the Scilab engine using Gateway

Scilab software architecture promotes a methodology to create external modules. Our concern here is to embed the capabilities of an external library to the Scilab engine. In Scilab, this is called *interfacing* [20]. Interfacing is the linking process, by which Scilab can use native function that is developed using a general purpose programming language, as a primitive Scilab function.

Gateway is a term used in Scilab for a function, written in C language, that is responsible for converting the data to and from the native function, and provides a call interface to the native function. It is also called *wrapper function* because it wraps the native function, so it can be regarded as a Scilab primitive function (Figure 1).

The gateway function relies on some header files, such as *api_scilab.h*, *MALLOC.h* and *Scierror.h*. These headers support the gateway with all function prototypes needed to interact with the Scilab engine.

Below is an example of a native function that is a part of the Scilab Networking Module. *UDPsend_dblData* sends data (doubles) using the UDP protocol. It is a pure C function that is called by the gateway.

As given in the following code listing, it opens a UDP socket for the Scilab process in order to send the data to the specified address and the port number.

```
int UDPsend_dblData(char *_stServerName,
int _iPort, double _dblData)
{
 int sock, length, n;
 struct sockaddr_in server;
 struct addrinfo hints,*res;
 char str[INET_ADDRSTRLEN];

 memset(&hints,0,sizeof hints);
 hints.ai_family=AF_INET;
 hints.ai_socktype=SOCK_DGRAM;

 if((n=getaddrinfo(_stServerName, NULL,
&hints, &res))!=0)
 fprintf(stderr, "getaddrinfo error:
%s\n",gai_strerror(n));

 sock = socket(AF_INET,SOCK_DGRAM,0);
 if (sock < 0)

perror("socket failed!");

 server.sin_addr=((struct
sockaddr_in*)res->ai_addr)->sin_addr;
 server.sin_family = AF_INET;
 server.sin_port = htons(_iPort);
 length=sizeof(struct sockaddr_in);

 /* send data using UDP */
 n=sendto(sock, &_dblData,
sizeof(double), 0, (const struct
sockaddr*)&server, length);
 if (n < 0){
 perror("UDP-send failed!");

printf("%s:%d",inet_ntop(AF_INET,&server.
sin_addr,str,sizeof(str)),server.sin_port
);}
 else
 printf("UDP-sending %f, n=%d\n",
_dblData, n);

 shutdown(sock,SHUT_RDWR);
}
```

Code 1. Native C function to send data using UDP.

Gateways incorporate various logical sections or steps. These steps will be presented as code listings for the example gateway that wraps the native function *UDPsend_dblData*. The first step, as presented in Code 2, is to declare the local variables that are required for the wrapper function. These local variables will be used by the Scilab engine for data exchange.

```
 int sci_udpclient(char *fname)
{
    SciErr sciErr;
    int* portAddr = NULL;
    int* dataAddr = NULL;
    int* domainAddr = NULL;
    double port = 0;
    char *domain = NULL;
    double dblData = 0;
    char* strData = NULL;
```

Code 2. Define local variables

In Code 3, we let Scilab check the number of input and output arguments which will be provided by the Scilab engine. Then, these input and output arguments are assigned to the local variables that have already been defined.

```
CheckInputArgument(pvApiCtx, 3, 3);
CheckOutputArgument(pvApiCtx, 0, 1);

sciErr =
getVarAddressFromPosition(pvApiCtx, 1,
&domainAddr);
    if(sciErr.iErr)
    {
           printError(&sciErr, 0);
           return 0;
    }
    sciErr =
getVarAddressFromPosition(pvApiCtx, 2,
&portAddr);
    if(sciErr.iErr)
    {
           printError(&sciErr, 0);
           return 0;
    }
    sciErr =
getVarAddressFromPosition(pvApiCtx, 3,
&dataAddr);
    if(sciErr.iErr)
    {
           printError(&sciErr, 0);
           return 0;
    }
```

Code 3. Define variables and addresses pointers

Code 4 presents the validity check for each parameter. The Scilab API provides various functions for

each datatype validation; here, we present only two of these functions: *isStringType* and *isDoubleType*. Respectively, these functions check whether the value given is of *string* type or of *double* type. When the validation is successful, it executes the function that is responsible for taking the data from the Scilab engine. For the local variables, we use *getAllocatedSingleString* to copy the string value to a local variable, and *getScalarDouble* to copy the double value to a local variable.

Notice that the first string value is for the domain address, and the second double value is for the port number.

```
/* ====== check inputs ======= */
    // check domain
    if(!isStringType(pvApiCtx,
domainAddr))
    {
            Scierror(999, _("%s: Wrong
type for input argument #%d: A string
expected.\n"), fname, 1);
    return 0;
    }
    if (
getAllocatedSingleString(pvApiCtx,
domainAddr, &domain) )
    {
            Scierror(999, _("%s: Wrong
size for input argument #%d: A scalar
expected.\n"), fname, 1);
            return 0;
    }
    // check port
    if(!isDoubleType(pvApiCtx, portAddr))
    {
            Scierror(999, _("%s: Wrong
type for input argument #%d: A Integer
expected.\n"), fname, 2);
    return 0;
    }
    if (getScalarDouble(pvApiCtx,
portAddr, &port) )
    {
            Scierror(999, _("%s: Wrong
size for input argument #%d: A scalar
expected.\n"), fname, 2);
            return 0;
    }
```

Code 4. Check and get parameters value from Scilab Console

Finally, in Code 5, we check the last parameter, which holds the data that we want to send, then call the native function that is responsible for sending the data using the address and the port number.

```
if (getScalarDouble(pvApiCtx, dataAddr,
&dblData))
{
    Scierror(999, _("%s: Wrong size for
input argument #%d: A scalar
expected.\n"), fname, 3);
                return 0;
}
UDPsend_dblData(domain,(int)port,
dblData);
sciprint("Data sent (UDP): %f size:
%d\n",dblData,sizeof(dblData));
```

Code 5. Calling the native function

3 Xcos Networking Palette

In this section, we will introduce an approach for developing a networking block library as Xcos palette. Rather than employing a native API, our approach adopts Scilab's native support for the Tool Command Language (TCL). TCL is a general purpose scripting language that was designed in the 1980s [21]. Scilab offers a native mechanism to invoke TCL language code directly from Scilab code using the *TCL_EvalStr* function.

TCL core supports TCP sockets, but UDP implementations have been provided as extensions of the TLC core [22]. This section presents a TCL-based implementation strategy that utilizes core TCP sockets. In Code 6, the *SERVER_open function*, which is used to open a new TCP socket on the given port number, is presented. This function just wraps several TCL code lines into Scilab code. The great advantage of this approach with TCL is that it is cross-platform, as all Scilab distributions contain TCL by default.

```
function SERVER_open(port)
 TCL_EvalStr("set ::SERVER_handle [socket
-server SERVER_newClient
"+string(port)+"]");
 TCL_EvalStr("fconfigure $::SERVER_handle
-blocking 0 -translation crlf;");
 printf("Server open for client Request at
Port%d\n ",port)
endfunction
```

Code 6. TCP server initialization

When developing external blocks for Xcos, *interface functions* are used to define the block appearance, the number of inputs and outputs and the behavior. Code 7 shows an interface function of the block that sends data to a remote client. The function has three argu-

ments: job, arg1 and arg2. Job specifies the mode of the Scilab block's interface function. There are three modes:
Plot: In this mode, data about the block are displayed on the plot.
Set: In this mode, the block has its parameters initialized using the *scicos_getvalue* function.
Define: In this mode, block appearance is initialized and also the inputs and outputs are checked.

```
function
[x,y,typ]=newcsi_block_m(job,arg1,arg2)

 x=[];
 y=[];
 typ=[];

 select job

 case "plot" then
standard_draw(arg1)
case "getinputs" then
[x,y,typ]=standard_inputs(arg1)
case "getoutputs" then
[x,y,typ]=standard_outputs(arg1)
case "getorigin" then
[x,y]=standard_origin(arg1)
 case "set" then

 x=arg1
 graphics=arg1.graphics;
 model=arg1.model;
 exprs=graphics.exprs

 while %t do

 [ok,ip,rv,exprs]=scicos_getvalue('Set TCP
server parameters',..
 ['TCP port';'no use';],..
 list('vec',1,'vec',1),..
 exprs)

 if ~ok then
 break,
 end
 model.rpar = [ip;rv];
 graphics.exprs = exprs
 x.graphics = graphics
 x.model = model
 break
 end

 case "define" then

 in=0
 out=1
 model=scicos_model()
 model.sim=list("mybf",5)
            model.out=1

            model.blocktype="c"
            model.dep_ut=[%t %t]
 model.label="TCP R"
            exprs=[string(["1234";"0"])]
            gr_i=['txt=[''TCP
RECEIVER''];';

     'xstringb(orig(1),orig(2),txt,sz(1),s
z(2),''fill'')']
            disp(gr_i);
 x=standard_define([4 2],model,exprs,gr_i)
 disp("ready to go")
 End

endfunctionsd
```

Code 7. The block interface function

C*omputation functions* are used to define the behavior of an Xcos block during a simulation. They have input parameters – block and flag. Block corresponds to the block which owns this function, and flag represents the simulation phase. Using the flag parameter, we can define the function behavior in different simulation phases of an Xcos model, such as initialization, simulation and termination.
In the following example, see Code 8, a computation function of the block which receives data over the network is given. Particular tasks are performed in different simulation phases. When flag has value 4 (initialization), the function starts the TCP server. If flag is 5 (termination), it stops the server. If flag is 1 (simulation), the computation function sets block output to what is received from the TCP server.

```
function block=mybf(block,flag)
 if flag==1 then
 block.outptr(1)(1)=retval;
 end

if flag==4 then
global retval;
retval=0;

 mode(0);

SERVER_close();
SERVER_open(block.rpar(1));

end
if flag==5 then
 SERVER_close();
end
```

```
 endfunction

function SERVER_readableEvent(str)

disp (str);
global retval;
retval=strtod(str);
endfunction
```

Code 8. A block computation function

4 Conclusion

After discussing the emerging requirements for having distributed simulation capabilities in model-based simulation environments for technical systems, the paper presents an implementation strategy for an initial peer-to-peer networking capability for Scilab and Xcos with the Scilab Networking Module and the Xcos Networking Palette.

The Scilab Networking Module exercises an implementation strategy that exploits the Scilab gateway, which enables interfacing with general purpose programming language APIs and the Xcos Networking Palette employs TCL as a general purpose scripting language. While utilizing general purpose programming language APIs is more complex, they provide flexibility for developers to extend the feature set of the API using general purpose programming languages. Running TCL scripts in Scilab, on the other hand, is easier to implement, but limits the user with the TCL capabilities.

This paper presents a first attempt towards a distributed simulation toolbox in Scilab. Future work includes developing a full capability implementation for peer-to-peer communication. Both TLC and Scilab gateway will be employed where appropriete. Further, we will be investigating the implementation strategies for shared bus architectures like HLA.

5 References

[1] S. L. Campbell, J. P. Chancelier and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos.* Springer, USA, 2006.

[2] C. Bunks, J-P. Chancelier, F. Delebecque, M. Goursat, R. Nikoukhah and S. Steer. *Engineering and scientific computing with Scilab.* Edited by Claude Gomez. Springer Science & Business Media, 2012.

[3] V. Hubka and W. Eder. *Theory of Technical Systems: A Total Concept Theory for Engineering Design.* Springer-Verlag, Germany, 1988.

[4] T. Xiao and W. Fan. *Modeling and Simulation Framework for Cyber Physical Systems.* Advanced Methods, Techniques, and Applications in Modeling and Simulation, Japan, Springer, pp. 105-115, 2012.

[5] E. Lee. *Cyber Physical Systems: Design Challenges.* 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), Orlando, FL, 2008

[6] K. Lyytinen and Y. Yoo. *Issues and Challenges in Ubiquitous computing*. Communications of the ACM, 45(12), pp. 63-65, 2002.

[7] J.C. Augusto and P. McCullagh. Ambient intelligence: Concepts and applications. *Computer Science and Information Systems,* 4(1), pp. 1-27, 2007.

[8] M.D. Petty. *Verification, Validation, and Accreditation*. Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains, Eds. Sokolowski, J.A. and Banks, C.M. John Wiley & Sons, 2010.

[9] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Proceedings of 2001 Winter Simulation Conference, Arlington, VA, 2001.

[10] O. Topcu, U. Durak, H. Oguztuzun and L. Yilmaz. *Distributed Simulation: Model-Driven Engineering Approach.* Springer, 2016.

[11] IEEE Standard for Modeling and Simulation High Level Architecture (HLA) – Framework and Rules. New York, NY, 2010.

[12] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification. New York, NY, 2010.

[13] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification. New York, NY, 2010.

[14] S. Pawletta, B.P. Lampe, W. Drewelow and T. Pawletta. *Eine HLA-Toolbox für Matlab.* Simulation und Visualisierung 2001 (SimVis 2001), Magdeburg, Germany, 2001.

[15] S. Pawletta and C. Stenzel. *Matlab(R) High Level Architecture Toolbox*. [Online] Available at: https://www.mb.hs-wismar.de/~stenzel/software/MatlabHLA.html [Accessed on 26 January, 2016].

[16] ForwardSim, *HLA Toolbox.* [Online] Available at: http://www.forwardsim.com/products/hla-toolbox/ [Accessed on 26 January, 2016].

[17] ForwardSim, *HLA Blockset*. [Online] Available at: http://www.forwardsim.com/products/hla-blockset/ [Accessed on 26 January, 2016].

[18] T. Theppaya, P.Tandayya and C. Jantaraprim. *Integrating the HLA RTI Services with Scilab.* 6. IEEE International Symposium on Cluster Computing and the Grid (CCGRID 06), Singapore, 2006.

[19] M.T. Jones. *BSD sockets programming from a multi-language perspective*. Charles River Media, Inc., 2003.

[20] *Scilab help.* [Online] Available at: https://help.scilab.org/docs/5.5.2/en_US/api_scilab.html [Accessed on 26 January, 2016]

[21] P. Raines and J. Tranter. *TCL/TK in a Nutshell.* O'Reilly Media, Inc., 1999.

[22] *UDP for Tcl*. [Online] Available at: http://wiki.tcl.tk/16733 [Accessed on 26 January, 2016].