Component Oriented Simulation Environment for Dynamic System Analysis and Testing

Sadek Bendago, Esam Elsheh

Abstract*— Computer simulation of dynamic systems lies at the center of engineering practice. Automatic control system design and testing is one of the fields where simulation programs are extensively used at various development phases. Older approach to build simulation programs was procedural one. Recently, the attention shifts toward Object Oriented (OO) approach. The main reason for this shift is to get benefits of reusability and adoptability as the main OO features. This paper examines an OO based modeling and simulation framework that could be used to construct various dynamic system simulations including real-time ones. Both adoptability and reusability are considered with paying attention also to extension of the number of simulation developers to include "old fashion" and "non-OO" programmers. To illustrate the usage of such an approach, a typical flight control system is used as an example. In this research paper the relevant functionalities of the flight control system are scattered among components. These components have the capability to interact to produce simulations.

Index Terms*— systems simulation; object oriented programming, flight simulation.

I. INTRODUCTION

Advance of programming tools caused simulation software developers to shift from functional decomposition development approach to object oriented one. In function decomposition a system is broken to sub-functions, while in object oriented approach [1] a system is broken down to entities (objects). Each entity should represent in natural way a real or conceptual element of the system. For example a Rigid Body is tangible element that has some properties like mass, inertia, etc. A Transfer Function as an entity is a conceptual element that could be used to represent system or subsystem dynamics. These superior features have encouraged many researchers to propose and design simulation packages based on the OO programming. For instance, in [2] flying simulation software is proposed based on OO according to the actual training of UAV (Unmanned Aerial Vehicle). In [3] the multi-agent and object-oriented technologies are used in designing UAV flight dynamic simulation. In the area of flight simulation, there are also several OO-based frameworks developed and adopted by famous simulation organizations. LaSRS++ framework [4] adopted by Langley Research Center (LaRC) is developed to be capable of supporting all real-time aircraft simulation at LaRC. The Aircraft System Simulation Environment and Tool-kit (ASSET) [6, 7] is another example of OO based environment. It is used for rapid flight prototyping. It is designed to be functionally decoupled and easy to learn. In this context, functional de-coupling refers to the design goal of isolating the software context in which models operate.

Revised Version Manuscript Received on December 12, 2015.

Saddek Bendago, Department of Information Technology, College of Engineering Technology–Janzour, Tripoli, Libya.

Esam Elsheh, Department of Information Technology, College of Engineering Technology–Janzour, Tripoli, Libya.

ASSET is targeted to run in a batch-mode as stand-alone or to be embedded into real-time simulation environment. ASSET by itself does not provide real-time capability. In both examples adoptability is approached via overriding featured in OO. Reusability is approached via abstracting sub-classes from super ones. From usability point of view, development using LaSRS++ or ASSET would require a considerable programming skills using Object Oriented Programming, preventing the developers who are not familiar with OO to use them directly. Adaptation to new variants of vehicles requires the access to source code at various levels in order to override object's default behavior. Reusability and adoptability are the major quality metrics used to evaluate simulation environments. These qualities are important to simulation organization, because they save development time and cut costs. Reusability is inheritable feature of OO. It allows multi-instancing of an element. For example many rigid bodies can be created based on a single RigidBody "blue-print" named Class. Adoptability is also OO inheritable, since some entities can be built based on other entities. For example AirVehicle could be constructed as extension of RigidBody entity. In this approach an attention is paid to usability also. This aims to allow developers with different profiles, to contribute directly in simulation environment. Better usability in this framework is approached by allowing users to define and manipulate complex models in the same way as variables used in procedural programming, despite the fact that OO technology is used to construct these models. The main motivation for this research is to enable one consistent simulation environment for analysis and test purposes, having in minds that the system development process includes a number of experts of different profiles and different levels of simulation skills and that it covers different phases, from early functional system simulations up to the complex simulation/testing procedures that include some parts of real system/process. As a result of these two facts, simulator development time is usually pretty long, some parts of job have been repeating many times, there are a lot of problems in building of new simulation experiments based on ones belonging to previous phases, etc. Component, Operator and Scripts are basic entities making up modeling and simulations environment in this approach.

II. PROCEDURE FOR PAPER SUBMISSION

Modeling of system dynamics starts with mathematical description of the system's behavior. Differential equations are used to describe it. A non-linear dynamic system could be represented mathematically as follows:

$$\vec{x} = \vec{f}(\vec{x}, \vec{u}) \quad (3.1)$$

Vector $\vec{x}$ is a set of variables that represent system's current state. Vector $\vec{u}$ is a set of variables that could influence system's dynamics (input set). At each simulation step, model (f) represented by equation (3.1) is manipulated to produce new state (3.2)

$$\vec{x}_{new} = \text{solve}(\vec{f}(\vec{x}, \vec{u})) \quad (3.2)$$

Analyzing such environment it is clear that it has three basic elements: model, state, and solver. Model element encapsulates aspects we interested in. State keeps information about current system state and inputs. Solver is process to manipulate system's model to produce systems state variables.

III. COMPONENT-BASED ENVIRONMENT STRUCTURE

Proposed framework to model and simulate dynamic systems is conceptualized around three basic notions. These notions are *Component*, *Operator* and *Script*. *Component* encapsulates dynamic and static features of a particular element in the system. The *Operator* encapsulates details to handle possible interaction a *Component* can have with other component(s). *Script* is textual form used to specify simulation interactions. Simulations are constructed by writing scripts. For example, a change of rigid body's mass is an interaction between both *Mass* and *RigidBody* components. Script 3.1 specifies this interaction.

// Script 3.1 --- RigidBody and mass interactions
Number Mass=1000;
RigidBody Aircraft
AirCraft $=_1$ mass

Predefined *RigidBody* and *Mass* components encapsulate properties and behaviors related to each of them. The Operator "$=_1$" has a predefined meaning. It encapsulates details of reaction of assigning new mass to it. Components could be designed and built by domain expertise at component level and used by system developers at simulation level. For example *RigidBody* could be built by physicists and used by aeronautical engineers.

IV. COMPONENT ANALYSIS AND DESIGN

At the beginning of component-based simulation development, components needed to construct a particular application have to be identified. Entity Attribute Analysis (EAA) technique used commonly in database design could be used to carry out this task [5]. In this technique a list of all possible elements or variables has to be generated, then grouping list's items into groups, each of these groups should represent a candidate component. For example in rigid body dynamics simulation, this list could contain items as (*mass, inertia, position, velocity, position of center of mass, etc.*). To specify the design of a component, there are two aspects that should be featured out. The first one is component's internal structure. The second one is component's interactions. Graphical based technique proposed by [5] could be used to specify component's internal structure. This graphical representation specifies both component's attributes and components that could interact with it. For example Figure 1. shows *RigidBody* component internal structure.

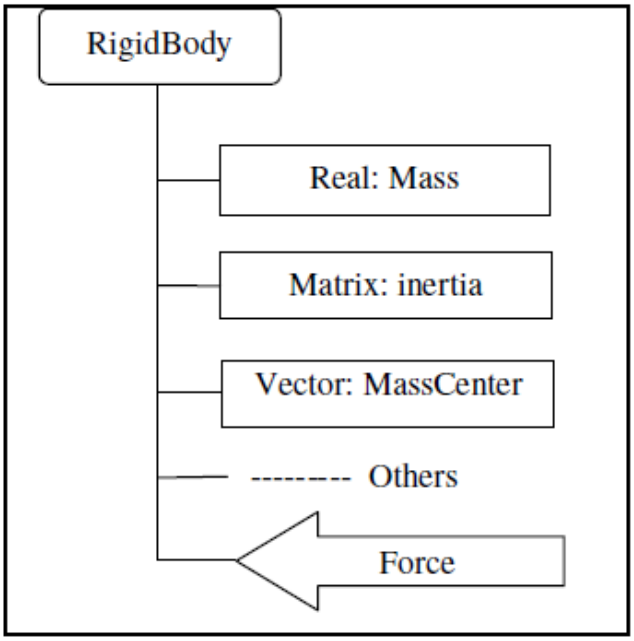

Figure 1. *RigidBody* internal structure

Rectangle shapes specify attributes of a component named *RigidBody* and their types. *Matrix* and *Vector* are also components themselves. *RigidBody* component should handle an interaction with *Force* component. Interaction rules could be specified in a tabular form. At each row an interaction is specified. A notation similar to Back Nour Forum (BNF) that is used for compilers syntax analysis is used here to specify an interaction. Table 1 samples interactions a *RigidBody* component could have.

Interaction	Description
[RigidBody] <RigidBody>=Force	Applying Force to Rigid Body
[RigidBody] <RigidBody>=Mass	Set New Mass for a RigidBody
Mass ::= Real Number	Specifies Mass as real float number

Table 1. Interactions a *RigidBody* can undertake

First row of Table 1 specifies that *RigidBody* can interact with *Force* component. This means that it should be able to handle such event. Having in mind these specifications, a rigid body dynamics simulation script could contain statements (interactions) as:

RigidBody Car
Force Engine
Car=Engine *//Force application*

V. SYSTEM DYNAMICS SIMULATOR (SDS++)

In order to illustrate usability of "component–based" concept represented in paper, a general framework to construct System Dynamics Simulator (SDS) is presented. This framework could model and simulate systems represented in equations (3.1 and 3.2). Applying analysis technique represented above, it becomes obvious that *Model*, *State* and *Solver* are major components of such environment. In the subsequent paragraphs design

of each of these components will be overlooked.

A. Model component

Model encapsulates information required to compute a dynamic system new state equation (3.2). Figure 2 shows *Model* component internal structure. *Script* property specifies model's mathematical description. This script could be implemented using primitive data types provided by customary programming languages. Application-based components could be used also to represent system's model.

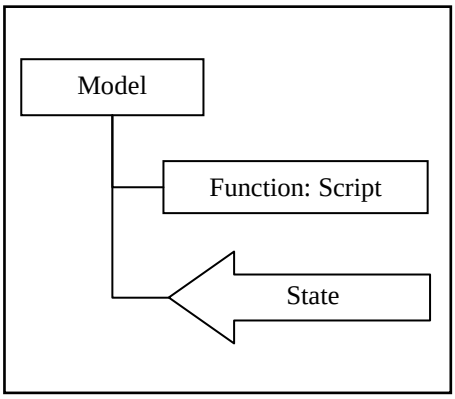

Figure 2. Model internal structure

For example, components like *RigidBody*, *Force* and *Actuator* could be used in a *Script* describing flight dynamics model. The existence of capability to use different levels of abstraction to describe system's model, would allow "old-fashion" programmer to contribute directly in simulation development, using their basic knowledge about system phenomena. They could also use components in the same fashion as ordinary primitive data type variables, despite the fact that OO approach is used to construct these components. Table 2 also specifies that *Model* component is able to interact with *State* component. SDS Interactions are another aspect that should be specified. It could be done applying BNF like notion. Table 2 highlights interactions that a *Model* can undertake.

Interaction	Description
<Model>=Script	Bind a script to a Model
[Vector] <Model> = State	Apply a state to a model
[Model] <Model> =State	Apply a state to a model

Table 2. Model component interactions

Name of a component that will be modified by the interaction is enclosed within "<>". The "[]" brackets specify component generated as a result of interaction process. For example in row three of table above *State* component acts on *Model* component, then an updated instance of *Model* is generated.

B. State

State encapsulates both sets of state and input variables ($\vec{x}, \vec{u}$, respectively). Based on equations (3.1,3.2) *State* should be provided with the capability to interact with *Vectors*. For example it should handle interactions like assignment of a vector, access to $\vec{x}, \vec{u}$ sub-vectors or assignment of single variable.

C. Solver Operator

In the context of dynamics system simulation the concept of solver is oriented toward numerical integration of differential equations representing system's model in order to compute new system state. In SDS Solver is conceptualized naturally as an extended-*Operator* of a *State* component. This *Operator* encapsulates procedure to compute model's new state. Table 3 shows definition of an extended State operator.

Interaction	Description
<State> =Model	Evaluate a model to compute it new state based on the current one.

Table 3. State's solver operator

As models and interactions specified above are implemented, general look of a *Script* simulating a dynamic system could appear as follows

```
// Script 6.1- Application specific model script
Function f(x,u)
Begin
   // Application specific model code. This code could use
   primitive-based or component-based data types
End
// Script 6.2 System Dynamic Simulation
Model DynSys
State  S (initial state)
DynSys=f
 do
  S=AirCraft=S
Untill End Of Simulaiton
```

Scripts 6.1 and 6.2 represent two levels of development. The first is an application level, where the developer concentrates on actual application modeling. The second one is simulation level. SDS components are glued together to construct simulation at this level.

D. SDS Implementation

Theoretically components specified above could be implemented using anyone of OO programming languages like Java, C++, etc. Components developed for SDS are implemented in C++ programming language (this is the reason of having ++ postfix). *Components* are implemented using *Classes* structures, while *Operators* are defined using operator overriding facility.

E. Interface Components

In many occasions dynamics simulators are connected to real or simulated devices, like actuators, sensor packages, etc. Usually models developers suffer from lack of knowledge about simulation environment's hardware details. Based on this natural fact, SDS++ is designed in a way allowing developer at simulation and application levels to define virtual interface variables and use them in the same fashion as "regular" ones. For this purpose signal generation and acquisition components are designed to facilitate interface mechanism. Figure 3 and Table 4 Summarizes *SigGen and SigAcq* components internal structure and interactions.

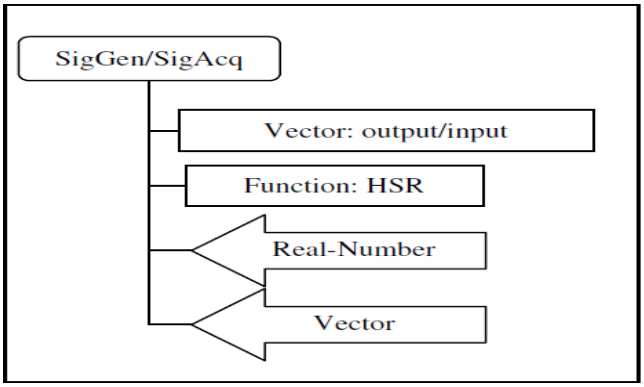

Figure 3. Interface components internal structure

From structure point of view, both *SigGen* and *SigAcq* are the same. The Hardware Service Routine (HSR) is pointer to procedure that encapsulates interface details, for example setting Digital to Analogue Converter (DAC) channel to some value(s). This routine also handles data representations and/or rescaling issues. *SigGen* and *SigAcq* interact with Real-Number and Vector components. Table 4 specifies these interactions.

Interaction	Description
<SigGen> = Real Number	Apply a single real number to signal generation virtual variable.
<SigGen>= Vector	Apply a set of real numbers to signal generation virtual variable
<RealNumber>=<SigAcq>	Read contents of signal acquisition variable into a real number
<Vector>= <SigAcq>	Read contents of signal acquisition variable into a set of real numbers

Table 4. Interface components interaction definition

As these operators are implemented, developers at simulation or model levels could define and use interface variables in the same way as ordinary ones. The following script illustrates such a feature:

SigGen DACChan0(HSRch0)
State S
……
DACChan0=S.x(7)

Script above simply sends contents of seventh state variable to a DAC channel zero. The *HSR* named *HSRch0* is a function containing details to handle value assignment.

F. Real – Time Components

When dynamic system simulator(s) is connected to real hardware device(s), such as sensors, actuators, etc, they ought to update their *State* at certain suitable frequency in order to emulate reality. SDS++ is equipped with a set of components to facilitate real-time features. Functionally, these components should allow developers to control the executions of their scripts. Real-time simulation of dynamic system is featured with:

1- Off line scheduling: the order of execution is known in advance. For example, typical sequence of executions in system dynamic simulation starts with reading sensors outputs, computing model's new state and generating output signals.
2- Resources as memory, inputs, etc., are defined before the simulation start. No dynamic allocation of resources is required.

Analysis, design and usage of real-time components could follow the same stereotype used for other components. *Task* and *Scheduler* are key components in SDS++ environment. They are designed to follow the same pattern as other SDS++ components. *Task* component holds the information about its start and stop time during the simulation time. It holds the time interval between consecutive task executions (Δt) also. It keeps also a pointer to *Task*'s actual code (*Script*), which has to be executed at every Δt interval. *Scheduler* component holds the information about set of tasks to be visited constantly in ordinal way. Upon visiting of each task, start and stop time has to be checked to decide if a task has to run or not. Both components accompanied with a set of operators that allow developers at simulation level to maintain and implement real-time constraints. At Simulation level, developers of a simulation code could organize these components in *Tasks* and *Scripts* as well as to set *Scheduler*. The following script shows a typical real-time system dynamics simulator.

```
Read()
  Begin
     SigAqc Input(HSRAdc)
    End

Process ()
    Begin
        State S
        Model f
          ….
    End
Out()
    Begin
     SigGen  Out(HSRDac)
      ...
  End
      // Main Script
Task      Read,process,Out
Main ()
    Schedular S(parameters)
    S=Read
    S+=Process
    S+=Out
    Do
      S=RTClock;
loop
```

Process, *Read* and *Out* are the tasks that scheduler should visit constantly. The main script starts with assignment of *Read*, *Process* and *Out* tasks to Scheduler *S* in sequence. This is the same sequence that will be used by *S*. In the main loop, each time *RTClock* is assigned to scheduler *S* the tasks list is visited and task illegible for execution is set to run. Practically assignment of real-time clock could be controlled via interrupts mechanism. Each time an interrupt signal is generated scheduler *S* is triggered to fetch his tasks list.

VI. COMPONENT-BASED FLIGHT SIMULATOR

The following is an illustration how SDS++ components could be tuned or customized to simulate flight dynamics In further step we illustrate other control elements and components that should be added to flight control system test environment. Finally, the extension consisting in including of a real orientation-sensor is discussed also.

A. Flight Dynamics Simulation

To model and simulate flight dynamics using SDS++, Model Script should be developed and then incorporated into SDS++ simulation level components.

B. Model Script

The model script shown here uses component-based approach. A set of components related to flight such as *RigidBody*, *Shape*, etc., are identified, analyzed and designed. Predefined interactions are also developed. Discussion of analysis and design of such components will be omitted here. Numerical data specifying "Sky-hawk" fighter aircraft are used. The following *Script* specifies component-based flight model:

```
// Model-level Script
FlightModel(x,u)
    Const double S=260.0, b=10.3, c=27.5, mass=570;
    RigidBody Skyhawk(f);
    // function f relates mass to ineria matrix
    Shape SkyhawkShape(g,S,b,c);
    // function g computes aerodynamic forces
    Force EngineF(0,0,thrust),AeroForce;
    Vector Xdot,parameter;
    Skyhawk=mass;
    AeroForce=SkyhawkShape=paramater;
    Skyhawk+=AeroForce+ EngineF
    RigidBody=!Shape
Xdot=Rigibody
```

In the case when a new variant of an air-vehicle should be modeled, functions f and g need to be revised. These functions could be implemented using procedural languages. This would allow "non-OO" experienced developers to contribute directly into flight model development.

C. State Model

Flight simulation maintains a vector $\vec{x}$ of twelve variables (components of linear and angular velocities, angular orientation and actual position in reference coordinate frame). Three-dimension *Vector* $\vec{u}$ representing control surface deflections is incorporated into the model also.

$$x = \{U, v, w, p, q, r, \emptyset, \theta, \psi, X_w, Y_w, Z_w\},$$
$$u = \{\delta e, \delta r, \delta a\}.$$

D. Solver Operator

Evaluation of a flight model to compute system's new state is encapsulated in "$=^1$" operator. Runge Kutta fourth order method is used for this purpose. However, the developers can override this operator with other solvers (methods). The following is the *Script* that manipulates above outlined components to construct non-linear non-real-time flight simulation.

```
// Simulation-level Script
main()
  Vector x,u
  State S=(x,u);
  Model AirJet(FlightModel);
 Do
  S=1AirJet=S
while (t < Stop)
```

E. Auto-Piloted Model

As flight model is completed it could be simulated as standalone model or it can be incorporated into the flight control testing facility. In this example we assume that flight model and controller model will run on two separate platforms. Information (signals) across these machines are interchanged via analogue interface (Figure 4).

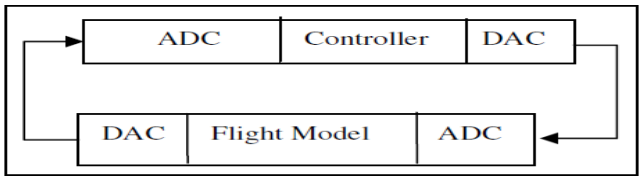

Figure 4. Flight control system test configuration

Previously developed components and scripts are used again together with some other components to develop real-time piloted simulator. *Actuator*, *Sensors* and *Controller* are new elements that should be added to simulation environment. Components representing these elements are designed and developed in the same manner used for other SDS++ components. There are two levels of abstraction that could be used to construct these components. At the first level, a particular component is constructed in order to encapsulate all its physical properties. For example a component named *Actuator* could be designed in order to provide possibility to write scripts as follows:

```
Actuator Elevator (parameter)
//define a variable representing an actuator
Actuator =Input
```

At the second level, more general component could be specified in order to model dynamics of these elements. For example, *Continuous Transfer Function* component could be used to model actuators, sensors, and controllers.

```
Polynomial  A,B
CTFucntion  Actuator(A,B);
Actuator=input
```

Signal generation and acquisition components are used also. Finally, scripts have to be organized in *Tasks* to be suitable for real-time environment components. In order to adopt flight control testing environment outlined above, *Interface* component for sending and receiving information via DAC and ADC respectively, should be added. Also *Real-time* components have to be set. Following list shows script written to run at Flight Model side. New added script lines are underlined.

```
// Script code at Flight Model side
  Vector x,u
  Polynomail A,B
  State S=(x,u);
  Model AirJet(FlightModel);
  CTFunction Elevator(A,B)
  SigGen  Out;
    SigAcq  In

// Main loop as a Task
```


Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

```
T1(){
        S.u[ELEVATOR]=Actuator=In
        S=[1]AirJet=S
        Out=S.x[PITCH]
}
 Task  T1(SimLoop)
 Scheduler  S+=T1
 Do
    S=t;
 While(t<Stop)
```

At the controller side, signal generation and acquisition components are used beside C*ontroller* component that could be implemented as continuous or discrete transfer function. Figure 5 shows flight model measured input and output.

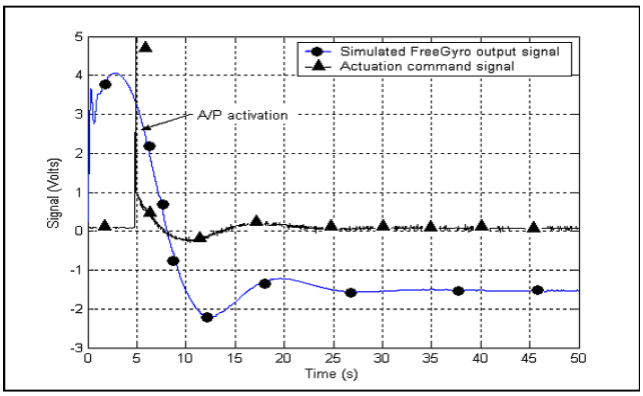

Figure 5. Simulator input/output signals

F. Hardware-in-the-Loop Simulation

At advanced stages of flight control system testing, real flight equipment and sub systems such as sensors and actuators are incorporated into the simulation scenario. In order to adopt this situation, software components representing these elements should be simply removed from the scripts above. Actual orientation sensors are usually used to check complete system. They are mounted on moving three-axis platform controlled by the output of flight model. The analysis of platform dynamics is of particular interest for control system designers. In this work we have added three-axis table (*3XTable*) component to allow inclusion of such dynamics. Variables representing this component can be defined as ordinary ones. If the real table is used, *3XTable* component is simply excluded. Figure 6 shows results produced by both *Flight Model* and *3XTable* Model.

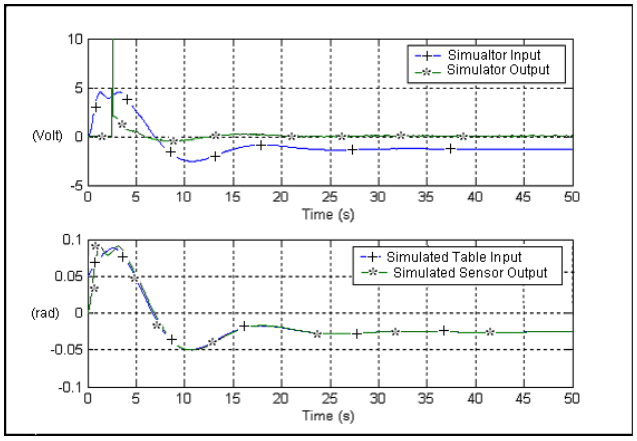

Figure 6. Inputs and outputs produced by Fl

Conclusion

Simulation environment used for design and testing of automatically controlled dynamic systems should be capable to satisfy a number of requirements. Reusability of system components' simulation models is the most important one, having in minds that a number of experts of different profiles take part in the overall design process, starting from early functional design analyses up to the simulations including parts of control system implemented as the live hardware units. Simulation methodology based on Object Oriented Programming technology was applied here, resulting in an environment (SDS++) useful for many dynamic system simulation scenarios. Flight control system simulation and testing have been used here as typical illustrative examples.

Two main ideas were the guidelines of this work. The first one was the requirement to reach such a level of abstraction that could fill the gap between experts familiar with physical aspects of system and simulator developers responsible for adequate system simulation/testing including hardware-in-the-loop simulation scenarios on one side, and to make it easy the steps from batch simulations to real-time ones, on the other. As a result of this, all simulator components are treated in a unified manner, independent on the fact whether they represent a part of real system or they are the support for real-time execution of simulation. Scripts specify the relationships between components (objects) while they remain the same whatever is the actual simulation scenario. It is something saving simulation software development efforts and time. The second important aspect of this approach consists in the attempt to fill the gap between the users familiar with procedural approach in simulation and the ones preferring OOT advantages. The system experts generally belong to the first group and they are allowed to generate their own models. After that, simulation experts belonging to the second group manipulate them. This feature is something allowing the safe usage of simulator by different types of users, at different levels of simulation.

REFERENCES

1. K.J. Hayhurst, and C. M Holloway, "Considering Object Oriented Technology in Aviation Applications," Digital Avionics Systems, 2003.
2. C. Yun and X. Li, "Design of UAV Flight Simulation Software Based on Simulation Training Method," WSEAS Trans. on Info. Science. and Applications, vol.10, no. 2, pp.37-46, 2013.
3. C. Yun and X. Li, "Research on UAV Flight Dynamic Simulation Model Based on Multi-Agent," Journal of Software, vol. 9, no. 1, pp. 121-128, 2014.
4. M. Madden, "Examining Reuse in LaSRS++-Based Projects," AIAA Modeling and Simulation Technologies, 2001.
5. S. Bendago, "Computer Simulation Environment for Flight Control System Development," PhD thesis, School of Electrical Engineering, University of Belgrade, 2004.
6. S. Derry and J.M. Maddalon, "Implementing Dynamic System Models in the ASSET Simulation Framework," AIAA Modeling and Simulation Technologies Conference, 2000.
7. P. S Kenney, "Rapid Prototyping of an Aircraft Model in an Object-Oriented Simulation," AIAA Modeling and Simulation Technologies Conference, 2003.