A Software-Defined Radio Simulation Method using Observer Patterns

N. A. Moseley and C. H. Slump
Signals and Systems group
EEMCS
University of Twente
Enschede
The Netherlands.

Abstract— **A problem with object-oriented simulation models is that internal model states are hidden and cannot be monitored easily. Object-oriented models are essentially black-box models. This article describes a method to expose the internal states of an object-oriented simulation model. Exposure of the states is achieved though application of the Observer software pattern in the form of data sources. Data sources can be connected to a data sink which then receives data from the sources. Connections between data sources and sinks are made though a broker. The globally accessible broker holds information on the available data sources.**

Some implementation details of a simulation framework based around the method are discussed. The framework is tested using a small simulation example on I/Q imbalance.

Although the focus is on software-defined radio and communication systems, the concepts presented here can also be applied to other types of object-oriented simulation.

Keywords— **OOP, simulation, communication systems**

I. Introduction

Simulating communication systems serves several purposes. Firstly, it is a way to develop and experiment with new ideas. Secondly, it is a tool to verify system behavior under controlled circumstances. The simulation of complete communication systems often involves many complex subsystems and processes such as error correction, channel estimation and synchronization.

Ideally, the designer can put together and simulate a complete communication system in very little time. Unfortunately, at present, this is not the case. Tools that allow rapid construction of simulation models, like Simulink, are not very efficient simulators. On the other hand, a direct implementation in an object-oriented programming (OOP) language such as C++ or Java can speed up simulations considerably but it takes more time to develop the simulation model. It is up to the designer to choose the appropriate method.

This paper focuses on the simulation of communication systems in C++. As previously stated, it takes more time to develop C++ models of communication systems. Luckily there exist several public-domain libraries to facilitate the development of these models. One such library is IT++ [1] which includes functions for OFDM modulation and demodulation, various standard channel models, FFT and many more useful tools. Using such a library can considerably speed up the development process.

Although the title of this paper suggests the presented method is only applicable to software-defined radios, any communication system simulation may benefit. In the case of software-defined radio, part of the simulation model **is** the radio. Thus, there is hardly any difference between a simulation model and the actual implementation. Simulation and implementation can be done using the same code base which guarantees a one-to-one relationship between the two.

It is assumed that the reader is capable of developing simulation models in an OOP environment and that he or she is familiar with the concepts of OOP.

II. OOP Simulation Models

Object-oriented simulation models are built in an hierarchical way. A complete system comprises subsystems which, in turn, may also consist of subsystems. An example of such a model is shown in Figure 1. It shows a digital phase-locked loop (PLL) built from an oscillator, a filter and a phase discriminator.

The PLL model has one input and one output signal. The output of the PLL is the signal generated by the oscillator. The goal of the PLL is to match the phase and frequency of its oscillator to the phase and frequency of the incoming signal.

In a real-world application, only the input and output signals are used. The PLL can be considered a *black box* as the internal signals cannot be accessed. When simulating such a PLL, the system designer

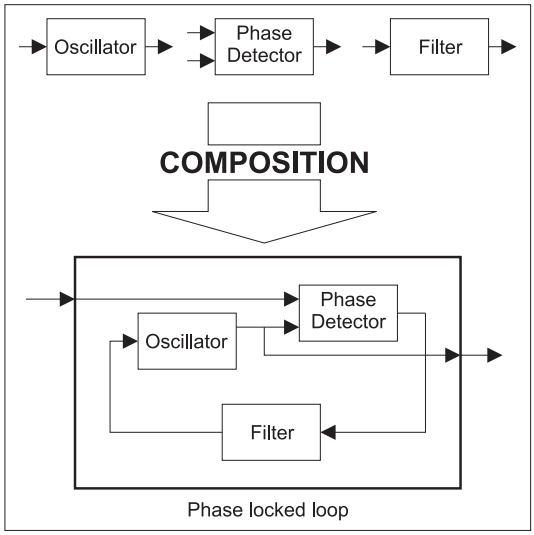


Fig. 1. Building composite objects from smaller objects.

might want to see the PLL's internal signals or *states*. The PLL should no longer be treated as a black box.

The OOP paradigm is well-suited to a black-box approach. Each model or submodel has its interface clearly defined. What goes on inside the model is not visible to the outside unless the model designer chooses to make some or all of the internal states visible. The standard way of exposing these internal states is through *get accessor* methods.

Get accessor methods are function calls that instruct the model to return a certain internal state. In case of the PLL one of the get methods could be called *'GetPhaseDetectorOutput'* which would return the voltage at the output of the phase detector. This approach is fine for simple models but it becomes cumbersome in cases where a model consists of multiple layers of submodels. Each time a new submodel is added, the designer must also add get methods to the layers above. Thus, using get methods is not a very efficient way of exposing the internal model states.

A second problem with get methods is that it is a *pull interface*. This means that the simulation framework must query, or *poll*, the model for its internal states very often. Each poll requires a certain amount of processor time. Thus, polling too often results in a slow simulation. But polling too little results in missing data.

A way around this problem is by using a modified version of the Observer software pattern [2].

III. The Observer pattern

The definition of the classic Observer pattern is: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and update automatically." [2]

The Observer pattern describes a way of monitoring data through a *push interface*. Each time the monitored data changes, a notification is sent to the monitoring object telling it that new data is available. In this way, polling is no longer required and no data is lost.

The Observer pattern only solves part of the problem. It defines how the internal states are monitored but not how the internal states are accessed. The internal states are still hidden from the outside and there is no way to monitor them. The internal states must be exposed.

IV. Exposing Internal Model States

The designer of the simulation model chooses which internal states to expose for monitoring by assigning each internal state a *data source* object. Each data source object can be connected to one or more *data sink* objects. In effect, the data source and data sink objects are implementations of the Observer pattern described earlier.

Whenever a data source produces data, the connected data sinks are notified of the new data available. The new data is transferred with each notification.

An example of an abstract simulation model is shown in Figure 2. The model in the figure contains two submodels and six *data sources*.

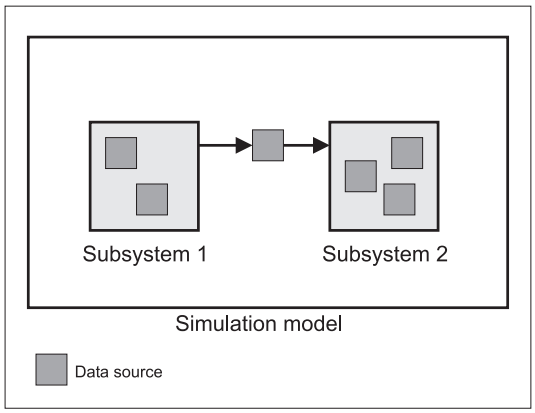


Fig. 2. A simulation model with two subsystems and six data sources.

A data source is given a unique human-readable name through which it can be identified. At the startup of the simulation, each data source registers itself with a globally accessible broker object, see Figure 3. The broker object serves as a directory of all the available data sources.

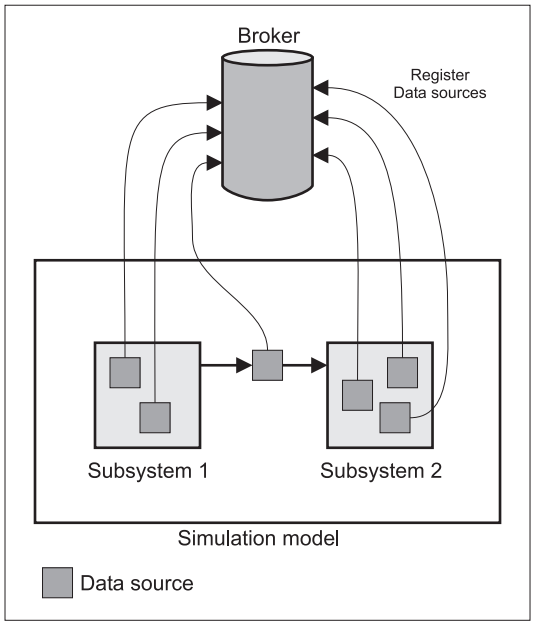


Fig. 3. A simulation model with two subsystems and six data sources.

The broker acts as a mediator in the source-to-sink connection process. A sink is connected to a source by invoking the broker's *ConnectSourceByName* function. This function is called with the source's unique name and the memory address of the data sink. The broker searches through the database to find the memory address of the source using its unique name. Then, the broker informs the source that a new sink has been connected.

An advantage of the data source method is that there are no changes to the interface of the simulation model like in the get accessor scheme. In case of software-defined radio this means that the designer only has to modify the data source's code to migrate from a simulation to an actual implementation. As the data source's code is located in a single source file, the transition from simulation to implementation is very easy.

V. Implementation details

The concepts mentioned above have been implemented in a small simulation framework. The framework was used for research published in [3]. This section discusses the implementation details of this framework.

A. Performing a simulation

A simulation starts with the registration of all the available sources with the broker. Then, the sources that must be monitored or logged are connected to sinks that process their data. The connections are made through the broker. The framework is now ready to start the simulation. A flowchart of a complete simulation in shown in Figure 4.

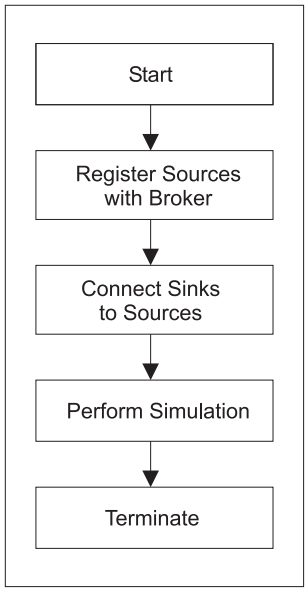


Fig. 4. Flowchart of a simulation.

During the simulation the sinks will receive data from the sources. The user can decide to write this data to disk or to display it on the screen. The data could also be further processed for representation.

B. Multithreading

One of the main advantages of the framework is that it allows the designer to split the simulation into a calculation, or number-crunching, part and a data storage and GUI[1] part. Each part runs in its own thread, see Figure 5. In this way, the simulation is able to make better use of a Hyperthreading [4] or multiprocessor system.

The simulation framework is built on the PThreads [5] library to support multi-threading. This library is available for Linux and Windows which enables the simulation framework to run on both platforms. The PThreads library supports *mutexes*, *semaphores* and *conditional variables*.

The main challenge of multi-threaded programming is inter-thread communication. The simulation framework solves this problem by including a multi-threading-safe data FIFO in each data sink. The data FIFO's are bounded; whenever a FIFO is full, the number-crunching thread is blocked until the FIFO can accept more data. The designer must make sure

[1]GUI stands for Graphical User Interface.

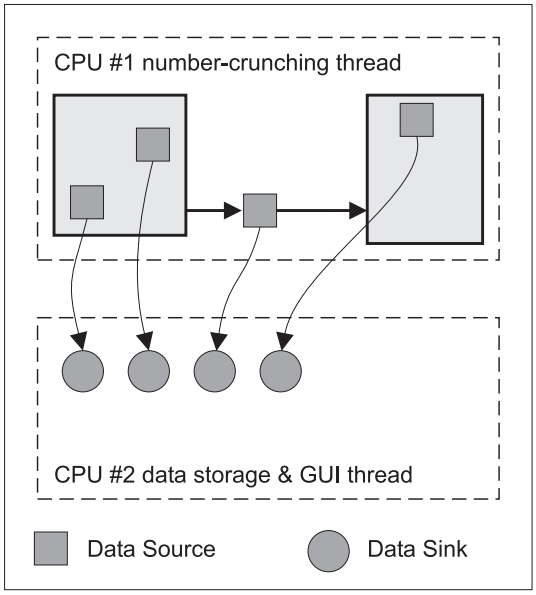


Fig. 5. Multithreaded partitioning of simulation.

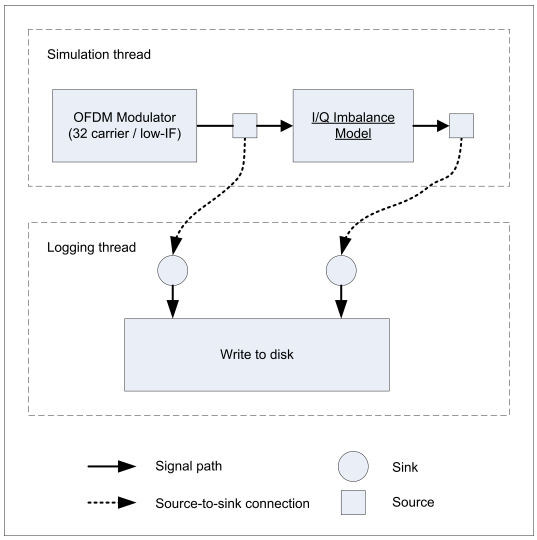


Fig. 6. Simulation setup of I/Q imbalance

that FIFOs in the data sinks are processed regularly otherwise the simulation will not run at full speed.

An advantage of this implementation of the framework is that sources are not required to be connected. If the data from a specific data source is not required, the data source can be left unconnected. It is even encouraged not to connect to such data sources because unconnected data sources introduce almost no overhead.

This feature allows a model to have tens or even hundreds of unconnected data sources without any significant speed penalty. The framework encourages the simulation model designer to implement as many data sources as expected to be useful.

VI. An example simulation

A simulation of receiver I/Q imbalance was done to verify the correct operation of the framework. The simulation produces values for the image-rejection ratio [6] as a function of LO phase imbalance. Verification of the framework is possible because the theoretically correct image-rejection ratio values are known.

A. Simulation setup

The simulation consists of a low-IF OFDM transmitter and an additional block for modelling I/Q imbalance at the receiver [6]. A block diagram of the simulation setup is shown in Figure 6.

The signal produced by the OFDM modulator is sent a data source and the I/Q model. The output of the I/Q model is routed to a second data source. Each data source is connected to its own data sink. The data sinks send the received signal data to a file on disk.

The simulation was run three times. Each with a different LO phase mismatch setting. The simulated settings were 5, 2 and 1 degrees of LO phase mismatch. The files generated by the simulation were read into MATLAB to generate plots.

B. Simulation results

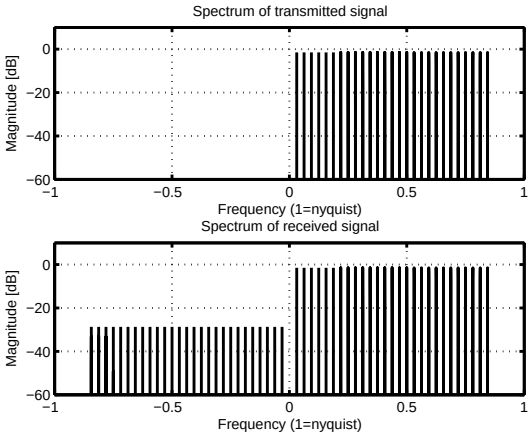
The spectra of the transmitted and received signals are shown in Figure 7. The image rejection ratio (IRR) can be read off from the received signal spectra. In this case, the IRR is the difference between the strength of the positive side of the spectrum minus the negative side.

The image rejection ratios from the simulation are in accordance with the theoretical values shown in Table I. The theoretical values from the table were calculated using the method from [6].

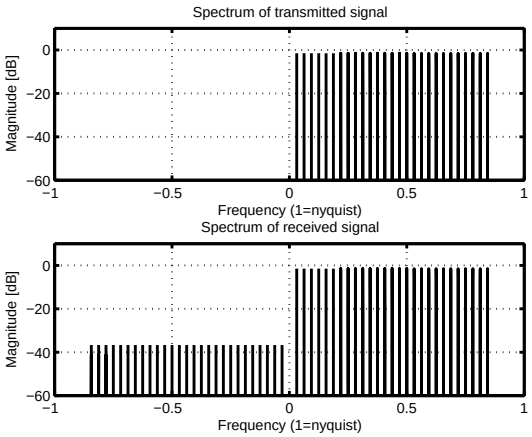
The simulation shows that the information emanating from the data sources arrives at the data sinks and is correctly written to disk. The method using the Observer pattern works.

TABLE I
Theoretical image rejection ratios.

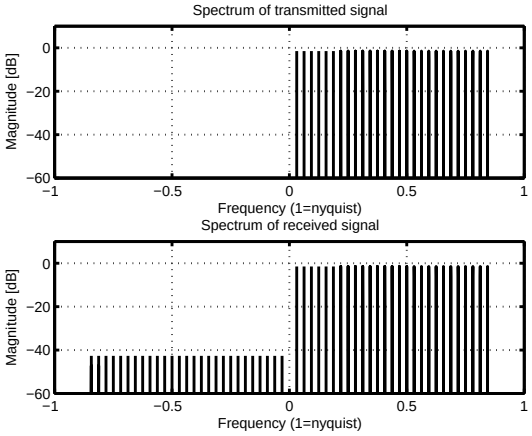
LO phase imbalance (degrees)	Image rejection (dB)
5	27.19
2	35.16
1	41.18



(a)Five degrees of LO phase mismatch



(b)Two degrees of LO phase mismatch



(c)One degree of LO phase mismatch

Fig. 7. TX and RX spectra under three different LO phase mismatch conditions.

VII. Conclusions

A method using Observer patterns was presented that allows access to internal states of a simulation model. The internal states are exposed though data sources. The data sources are connected to data sinks which, in turn, post-process the data or log the data to disk. Connections between data sources and data sinks are made though a broker. The broker holds the human-readable names of the available data sources.

The method was implemented in a simulation framework. The framework was used to perform an I/Q imbalance simulation. It was also used to do the simulation in [3]. The described Observer-based method was successfully applied in both simulations.

References

[1] "IT++ math and signal processing library," http://itpp.sourceforge.net/latest/, 2005.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison Wesley, 1995, ch. 5, pp. 293–304.

[3] N. A. Moseley, R. Schiphorst, and C. H. Slump, "The effects of non-linear power amplifiers on the performance of a hiperlan/2 system," in *DSPenabledRADIO Proceedings*, University of Southampton. IEE, September 2005.

[4] Intel, "Hyperthreading website," http://www.intel.com/technology/hyperthread/.

[5] B. Nichols, D. Buttlar, and J. P. Farrell, *PThreads Programming*, 1st ed. O'Reilly, September 1996.

[6] M. Valkama, M. Renfors, and V. Koivunen, "Advanced Methods for I/Q Imbalance Compensation in Communication Receivers," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2335 – 2344, October 2001.