Single and Multi-Station Virtual Simulation Design Patterns

Douglas D. Hodson¹, David P. Gehl²

¹Air Force Institute of Technology, WPAFB, OH, USA
²L-3 Communications, Link Simulation & Training, Dayton, OH, USA
Email: douglas.hodson@afit.edu, david.gehl@l-3com.com

Abstract—In a virtual simulation, people and real system hardware interact with a simulated system. Introducing these real-world elements into the simulation environment imposes timing constraints which, from a software standpoint, places the design into the class of a real-time system. Considering these requirements, we present several software design patterns appropriate for the development of single- and multi-station real-time virtual simulations. A variant of the model-view-controller architectural pattern is introduced followed by the development of a supporting component pattern that facilitates the development of single-station hierarchical simulation models, graphical displays, and network input/output (I/O) that need to meet real-time constraints. These patterns are extended to system designs that include multiple PCs with several graphics cards to support the development of multi-station virtual simulations (i.e., simulators that include multiple people or operators).

Keywords: Distributed Simulation Real-Time

1. Introduction

Virtual simulations involve people or real system hardware interacting with a simulated system. In either case, the software system (the simulation) interfaces and interacts with driving functions (input signals) [1] generated by a person or hardware component and responds by producing outputs. For a typical flight simulator, interaction includes input from stick and throttle devices and output in the form of graphical displays. Software systems designed to meet latency requirements due to these real-world interactions fall into the class of real-time systems. The organization of real-time systems and the quantitative methods used to evaluate particular designs can be found here [2], [3].

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design [4]. It is a description or template for how to solve a problem in many different situations. We are particularly interested in tailoring the Model-View-Controller (MVC) and the Component design patterns to the domain of virtual simulation. The MVC pattern provides a high-level architectural structure of an application and classifies objects according to the roles they play. The Component pattern is used as a basis to implement those specific objects.

We incorporate accepted real-time software organization paradigms into these patterns so that quantitative methods can be used to estimate the performance of virtual simulation applications. Incorporated paradigms include the separation of software code into foreground and background tasks while the scheduling of individual jobs (i.e., software code) mimics a fixed cyclic scheduler. The patterns also incorporate hierarchical modeling concepts to define modeled systems.

For each pattern, we assume an implementation that leverages modern object-oriented software techniques. This provides the flexibility to conveniently utilize the concepts of “selective abstraction” and “focused fidelity” to prune object trees, thereby improving system performance.

2. Model-View-Controller (MVC)

In the MVC pattern, there are three types of objects: model objects, view objects, and controller objects. Figure 1 shows the roles these objects play in the application and their lines of communication. When designing an application, choosing or creating custom classes for objects that fall into one of these three groups is a major step since it determines boundaries and communication with other types of objects occurs across those boundaries [5].

For a particular application domain, Model objects represent special knowledge and expertise; they hold an application’s data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects and, ideally, a model object has no explicit connection to the user interface [5].

For a virtual simulation application, the model object is the simulation itself. It contains all simulation state data, behaviors in the form of hierarchical system models, and manages time advancement. The model object as defined in the MVC pattern should not be confused with simulated system models.
A view object knows how to display or present data to an external viewer. The view is not responsible for storing the data it is displaying and comes in many different varieties. For a virtual simulation, the view includes the drawing of graphical displays such as GUI interfaces and interactive operator displays. In the case of distributed virtual simulations (DVS) or Live-Virtual-Constructive (LVC) simulations, a view is responsible for sharing simulation state data across a network to other interconnected simulation applications.

The controller object acts as the intermediary between the application’s view objects and its model objects. Ideally, it is the sole source of user inputs and connects the simulation to its graphical displays. Practically, one often merges the roles played by an object. For example, an object that fulfills the roles of both controller and view is called a “view-controller” [5].

![Simulation Design Pattern](image)

Fig. 2: Simulation Design Pattern

A view-controller is view layer centric. It “owns” the views, while still managing the interface and communications with the model. Combining roles like this is common [5] and is reflected in our tailored MVC design pattern for virtual simulations as shown in Figure 2.

The simulation pattern in Figure 2 consists of a top-level “Station” object containing one simulation object (i.e., the model in the MVC pattern) and multiple view-controllers. We call this top-level object a Station to reflect its close association between the management of I/O functions and visual displays which is what real operators interact with. The Station object also manages high-level functions for creating the threads associated with each of the view-controllers, as needed.

The simulation object consists of a list of players which are assembled as a set of hierarchical-based system models consisting of systems with sub-systems. The simulation object manages simulation time and provides features needed to implement a fixed cyclic scheduler. The view-controllers consist of handlers that read and/or write to I/O devices, interactive graphical displays and interoperability network interfaces. The network interoperability interface for sharing simulation state data supports concrete implementations of a variety of standards such as the Distributed Interactive Simulation (DIS) [6], the High-Level Architecture (HLA) [7], and the Test and Training Enabling Architecture (TENA) [8] specifications.

2.1 Multi-Threading

Ideally, the execution of a simulation application based upon the MVC simulation pattern would consist of a loop that would sequentially read inputs, execute the system models, and generate outputs (i.e., update graphics and process network activities) once per frame. This is an acceptable execution strategy for constructive simulations, where the requirement to execute in real-time (i.e., in sync with wallclock time) is often relaxed since everything is simulated by models. But a virtual simulation that performs all of these tasks in real-time, however, is limited by processing power, So this approach becomes problematic as frame rates increase, thereby reducing the amount of time to complete all tasks.

To resolve this fundamental problem, the processing time associated with input devices, graphic display(s) and interoperability network management functions (i.e., the view-controllers) can be partitioned into separate periodic tasks, each executed asynchronously with respect to each other, at particular frequencies. For example, the update rate associated with graphical displays might be much less than the rate at which the simulation advances time. Furthermore, the division of software code into foreground and background tasks reduces the workload associated with processing time-critical tasks. The challenge is to organize software code to promote this separation of work. This is one of the central goals of our tailored Component design pattern.

3. Component Pattern

The simulation pattern, as shown in Figure 2, is the first step towards separating a virtual simulation application into high-level objects that can be executed independently. Further improvement can be made by partitioning the real-time and non-real-time jobs defined by those independent objects (i.e., the simulation and view-controller objects) into foreground and background tasks. We introduce a Component design pattern which facilitates this separation while simultaneously supporting hierarchical modeling.

---

1For an object-oriented system, a “job” is a code fragment to be executed (e.g. a function or method call).
As most systems selected for simulation-based analysis are complex [9], and because managing the complexity of models is a challenging task, large systems are seldom modeled in a monolithic fashion. In fact, they are usually divided into smaller, interacting subsystems. The subsystems themselves are further divided into smaller sub-subsystems until a manageable level of complexity is reached. In other words, the *system under study* can be viewed as a “system of systems”. This subdivision of a system results in a hierarchical organization of the *system under study* itself.

Fig. 3: Hierarchical Player Model

An example hierarchy is shown in Figure 3, where the top level model is a “player” or “entity” within the simulation. The player is composed of both a dynamics and a sensor model. The sensor model is a composite of several sensors, namely, radio frequency (RF), infrared (IR), and electro-optical (EO) models. Dynamics is composed of an aerodynamics and propulsion model.

Hierarchical models from a software engineering point of view are software “components.” Conceptually, a component is an entity, a real-world object, that is viewed as a “black box.” Its interface, behavior, and functionality are visible but its implementation is not [10]. Components naturally map to object-oriented implementation paradigms supported by languages such as C++.

Gamma [4] contains a catalog of commonly used design patterns in software development and provides solutions developed and evolved over time. *Structural design patterns* provide classes and objects that form larger structures. Of particular interest for hierarchical modeling is the *composite* pattern in Figure 4 which implements hierarchical models in object-oriented programming languages.

The composite pattern uses a tree structure where components can have *children*, i.e., sub-systems and sub-sub-systems. The *Component* class declares the interface for objects in the composition and implements default behaviors for all the classes. The *Leaf* class has no children while the *Composite* class defines behavior for components that have children. The *operation* method is a placeholder for the functionality of the model. Using this structure, modeled systems can be divided into sub-systems and defined as *Components* through inheritance.

When implementing a composite pattern there are trade-offs related to software design safety and transparency. Gamma provides an extensive discussion that considers several implementation approaches. For example, the component class declares the *add* and *remove* methods to provide a transparent interface for all components, but these do not make sense for a leaf. We consider these trade-offs as we adapt this pattern to the domain of system modeling and real-time processing.

The hierarchical-based approach addresses model complexity, but does not address the temporal performance of code execution, specifically, the reliable completion of jobs at or before their deadline. We recommend partitioning of code into real-time foreground and non-real-time background tasks.

Given hierarchy models with the structural composite patterns shown in Figure 4, software partitioning can be incorporated by replacing the single *operation* method by two methods, *updateTC*, and *updateData* as shown in Figure 5. The *updateTC* method (where TC means time critical) is a placeholder to implement a real-time task which includes calculations associated with updating model state space. Less time-critical jobs, such as saving or logging data to a hard drive is placed within the *updateData* method.

We add and explicitly pass the simulation step-size (sometimes referred to as *delta-time*) as a parameter. Step-size is used by mathematical calculations associated with system models. Since *updateTC* automatically calls all of its children’s *updateTC* methods, executing a complete hierarchical model (implemented as a component tree) occurs with a single method call to the root component.

Our component design pattern considers all components to be composites. In other words, when modeling systems, sub-system, and sub-sub systems, there are no leaves, as each
model is an abstraction at some level.

Consider, for example, the player model in Figure 3. To implement this system, several models are created by subclassing from the Component class as shown in Figure 6. Component models whose functionality is described by a set of differential equations might include a numerical solver in the updateTC method. Other background, less time critical jobs, such as saving vehicle position data at each simulation step for analysis, is in the updateData method. After each component model is built, the complete flight control system is assembled into a component tree that is the complete modeled system. Subsequent execution or simulation of the modeled system occurs by calling the updateTC method of the root component.

### 3.1 Scheduling Code Execution

Designing a software system to meet temporal requirements is a scheduling problem. More formally, to meet a program’s temporal requirements in real-time systems, a strategy is needed for ordering the use of system resources [3]. This strategy results in a schedule for executing jobs. We are particularly interested in how to schedule jobs to maintain a consistent simulation state space.

To accomplish this, we design a cyclic scheduler which specifies when jobs are executed. The schedule is static, which may not be optimal, but is highly predictable and simple to implement. A cyclic scheduler makes decisions periodically. The time interval between scheduling decision points are called frames (F). Scheduling decisions are made at the beginning of every frame and there is no preemption within a frame.

A notional structure for a scheduler is shown in Figure 7. Frames are grouped into a “cycle,” and subdivided into an arbitrary number of phases. Frames are divided into phases to resolve data and control dependencies among jobs and specify an execution order. Adding features to support static scheduling in our Component class is as simple as adding attributes, specifically, cycle, frame and phase attributes in the form of class variables as shown in Figure 8. Subclasses of Component can be built that not only partition model code (i.e., jobs) for execution in the foreground and background, but explicitly define which frames and phases jobs should be processed. Providing direct access to scheduling attributes allows the developer to design a model or set of models that balances execution load across frames.

Consider a system model derived from the Component class with the updateTC method coded below:

```c
class Component {
public:
    unsigned int cycle;
    unsigned int frame;
    unsigned int phase;

    void updateTC(float dt) {
        switch (phase) {
            case 0: // update position
                break;
            case 1: // process radar interactions
                break;
        }
    }
}
```
The phase attribute is used to impose an execution order within each frame for modeling systems. Conditional code before the switch statement can be inserted to limit processing to selected frames within a cycle. A very common technique conditionally selects a single frame, all even or odd frames, or all frames within a cycle for execution. The parameter “dt” (delta time) is the simulation time advance step-size and is passed to the updateTC method and made available for system model calculations.

3.2 Modeling a Player

Consider a player or entity defined by an object tree specified by the set of Components instances \( \{C_1, C_2, C_3, ..., C_k\} \). The cyclic scheduler for the object tree has \( p \) phases, and \( f \) frames per cycle. The maximum execution time for the task defined by this single hierarchical system model can be determined by computing the execution time in each frame,

\[
e_f = \sum_{comp} \sum_{phase} e_{f,c,p},
\]

where \( comp \) is the set of components, followed by selecting the maximum frame execution time in the cycle,

\[
e_{player} = \max_{1 \leq f \leq cycle} \{e_f\}.
\]

For a virtual simulation, this is the execution time of a single instance of a player managed by the simulation object shown in Figure 2. Specific application of rate monotonic quantitative methods [11] for a single PC architecture can be found here [12].

3.3 Graphics and Input/Output

To support unique features of view-controller objects, specialized Components can be created with additional methods. For example, just as the single operation method in Component was replaced with updateTC and updateData to partition jobs, additional methods can be added to support the execution of specific jobs unique to a particular view-controller. Effectively, each new method defines an independent execution path through a hierarchical system model or object tree.

Analogous to the updateTC method provided by Component, the Graphic class provides a draw method for specifying graphic operations. In a similar vein, the NetworkIO class provides two methods for receiving and transmitting state data across a network, inputFrame and outputFrame. The NetworkIO class also serves as an abstract interface to support a wide range of interoperability protocols providing a clear separation between models and specific interoperability implementations.

Since the Graphic and NetworkIO classes are specialized Components, they can use updateTC for real-time model execution and updateData for background processing. For example, Graphic-based components can use updateTC, graphic operations in draw, and non-real-time background processing in updateData. Strictly speaking, this violates the spirit of the MVC pattern as the model would be closely coupled with the view and controller, but is acceptable to meet temporal constraints.

4. System Abstraction

Implementing hierarchical, component-based models using the Component design pattern efficiently implements “selective abstraction.” Selective abstraction [13] reduces the complexity of models by identifying and discarding details of the model which have minimal impact on the overall results. This allows the developer to prune the object tree at selected points to reduce the level of complexity to improve runtime performance.

Another approach starts with highly abstract system representations and adds fidelity as needed. We introduce the term “focused fidelity” to capture this concept. Focused fidelity provides the appropriate level of detail (resolution) to the system under study to provide the required accuracy while eliminating undesirable system inputs. This is important because complex models that are not directly under study can affect independent variables, which are inputs into the system under study, and can therefore confound the study results. Additionally, it is inefficient and often counterproductive to develop more complex models than needed for the simulation.

Our Component class provides the means to implement selective abstraction and focused fidelity concepts. Applying them reduces simulation development time and cost, while simultaneously improving runtime performance and validity of simulation results.

5. Multi-Station Simulator Patterns

The simulation design pattern presented in Figure 2 is most frequently used to develop single-station simulators (i.e., simulators that contain a single person with a limited set of graphical displays). These simulators consist of a single model (i.e., the simulation) and multiple view-controllers executing on the same PC. As organized, this approach takes
full advantage of multi-core, multi-CPU PC-based architectures. If multiple single-station simulators are assembled to create a DVS or an LVC, each application creates their own simulation object and shares state data through a common interoperability interface.

We now consider multi-station simulators which often consist of many more graphical displays than a single PC or a single graphics card can drive. This is a common situation for simulators that include multiple people each with their own set of interactive graphical displays. This case is different than the organization of a standard DVS or LVC because there is a single simulation object.

The design considered consists of a single simulation “executive” with scalable well-defined interfaces to support the data distribution and change request requirements from multiple PC’s that are responsible for providing their own set of interactive graphical displays. To support this requirement, we extend the simulation pattern through the introduction of a Interface class which provides another level of indirection between station graphics (i.e., view-controller(s)) and the simulation object (i.e., model).

As an example, Figure 10 shows an apparatus that can be used to fully test graphical displays independent of the simulation. As shown, a graphical display is associated with an interface object (Interface::System) which is responsible for setting and getting data associated with a particular modeled system. A specialized version of this interface (i.e., Test::System) serves as a surrogate simulation for testing purposes.

Figures 11, 12 and 13, graphically depicts the design patterns associated with the logical development of a multiple PC, multi-station simulator. Figure 11 provides a different application of the Interface class on a single PC by associating the specialized version of the interface (Local::System) with one of the simulated Player system models. Figure 12 effectively implements the test apparatus shown in Figure 10 across multiple PCs. To support this, a server (Server::System) interacts through a network infrastructure with a client (Client::System) to provide data distribution and change request services. Finally, Figure 13 shows the implementation of a full system design that includes multiple computers – a single “simulation executive” connected to multiple PCs each drawing interactive graphical displays.

6. Final Thoughts

The patterns presented have not been developed in isolation. In fact, they have been carefully crafted and used for many years by simulation engineers. They are heavily used by the open-source OPENEAAGLES [14] framework. Embracing these design patterns promotes good software designs that consider real-time requirements and leverage multiple PCs to develop multi-station simulators.

The views expressed in this article are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the US Government.

References

Fig. 12: Multiple PCs Test Apparatus

Fig. 13: Multiple PCs with Simulation Executive


