CBA Design Overview Technical Brief

August 18, 2011

ZedaSoft, Inc.
2310 Gravel Dr
Fort Worth TX 76118
817-616-1000
www.zedasoft.com

Introduction

The Container-Based Architecture for Simulation (CBA) is a family of object-oriented software frameworks that together implement ZedaSoft's simulation system design "A Container-based Architecture for Simulation of Entities in a Time Domain" (U.S. patent 7,516,052).

CBA is a general system for managing the update of arbitrary data over time. It makes no assumptions about the data being managed; instead, it provides an object assembly model, execution life-cycle management, network distribution, and other general services that can be used to model specific problem domains in a flexible way.

The ideas behind CBA were initially developed in 2002. At that time each member of the design team had worked in the simulation industry for 20 years or more, and each had extensive experience with object-oriented development platforms outside of simulation. We looked at the design and development of CBA as an opportunity to combine our experiences from both of these worlds into a better simulation framework.

Sidebar: Object-Oriented Design Terminology

CBA is an object-oriented system. There are many concepts in object-oriented design, and we mention a few of them frequently in this paper. For the uninitiated reader we'll establish some definitions:

Class – a collection of data and executable code that together defines a model of a concept or real-world thing. Classes can inherit data and executable code from other classes. A class from which other classes inherit is called a "superclass" of those classes. Classes that inherit from another class are said to be "subclasses" of that class. Classes can be aggregated hierarchically to build up complex models. Classes usually must be "instantiated" to produce executable things, although we often define executable operations that can be requested of the class itself.

Instance – when we talk about "objects", we're really talking about instances of classes. A class can be thought of as a template for executable components that might be created, and instances can be thought of as the created components. Zero or more instances of a given class can be created provided the class doesn't suppress the creation of instances. The process of creating an instance from a class is called “instantiation”.

Method – methods (also called "member functions" depending on who you're talking to) are executable operations that can be performed against either a class or an instance.

Interface – an interface defines a set of methods without defining their implementation. Interfaces can be implemented by zero or more classes. Interfaces allow us to build "loosely coupled" systems. Such systems minimize explicit dependencies between classes and make it easier to define extension points for third-party users of a system.

Motivations

Before starting on the CBA design, we spent some time identifying shortcomings of the technical approaches we had encountered while working with legacy simulation systems. Some of the ones that stood out included:

- Simulators tended to be "ownship-centric". To build a new simulator somebody rolled a cockpit shell complete with avionics displays into a room and told us to make it go. All subsequent software development work revolved around the cockpit; the cockpit *was* the simulator, and the software was just a peripheral resource that catered to that particular cockpit's needs.
- The software commonly placed hard-coded restrictions on the number of components that could participate in the simulation. If the software systems were moved to a more powerful computer they still couldn't support bigger simulations without modification.
- Applications were hard-linked and impossible for end users to extend.
- Graphics presentations and generation of the data they consumed were intertwined. In extreme cases the entire simulation – core algorithms, cockpit graphics, out-the-window scene, all of it – might be implemented in just one or two source files. This made it impossible to have more than one view of the activities in the simulation.
- Components were generally difficult to separate. This presented challenges when a code base had to be prepared for export, or unclassified components had to be swapped out for their classified counterparts inside a closed area. The "solution" usually was to make a full copy of the code base, then make one-way modifications. When improvements were made in the original code base there was now no easy way to get those changes into any derived code bases. The process of modifying a code base for export or closed-area use was also time-consuming and expensive.
- Simulator code tended to be locked in to a specific platform and/or computer configuration.

Eliminating these shortcomings led to the main design goals of the system:

- *The system must use a peer approach.*

  We decided that the ownship-centric approach to simulation was backwards and restrictive. The cockpit should be nothing more than a station where displays could be realized and one or more human operators could interact with multiple virtual entities. The "ownship" might be the most sophisticated entity in a simulation but otherwise should not enjoy any more privileges than the other objects in the virtual environment.

- *The system must eliminate limits on scale.*

  The legacy simulations with which our team had previously been involved typically used fixed-length arrays to model sets of entities. A side effect of this approach was that the maximum number of entities that could be modeled was defined at the time the system was built. For example, there might be a limit of 20 air threats, or 5 bombs in flight simultaneously. The design must be such that the processing potential of the simulation host computer and network were the only barriers to scale.

- *The system must be object-oriented.*

  The simulation problem domain is a natural fit for object-oriented design at almost every level of detail. At the highest level the simulation is a collection of entities all operating together in some virtual environment. Each entity is another finer-grained aggregation. For example, an aircraft is composed of a motion model, engines, fuel system, avionics, possibly a virtual pilot, and anything else needed to complete the model. Some of these systems may be further subdivided, such as modeling the individual tanks in a fuel system. If you back far enough away from the details it's easy to see that there is a general containment pattern that repeats at each level of detail to form a "tree" of objects.

- *The system must be loosely coupled.*

  The design must have no explicit class dependencies that might defeat flexibility. Interfaces should be used wherever possible as a basis for the interaction between objects.

- *The system must be extensible without requiring modification of the core system.*

  We must be able to introduce new types of entities without having to make changes to the core system. This would enable end users to add their own classes that could participate in the running simulation without the core system needing to change to accommodate the additions.

- *The system must support a multi-viewer capability.*

  When we abandoned ownship-centric thinking, and instead started thinking of remote views as nothing more than a display and interaction point for the objects in the simulation, it opened up the possibility for zero or more stations to be in operation simultaneously for multiple entities. This included multiple simultaneous connections to a specific entity. This required clear separation between data generation and consumption of that data by external view applications.

- *The system must promote cross-platform executable component reuse.*

  A great deal of time and effort has been spent in the simulation community inventing ways to exchange data between simulation silos, DIS and HLA being two obvious examples. The design must support executable component reuse in addition to data reuse. It should be possible to maintain a depot of ready-to-use classes that can be downloaded and instantiated at the time a simulation scenario is created or executed.

- *The system must promote simulator execution on a variety of hardware platforms.*

  We should be able to run a simulation on anything from a laptop to a large multi-computer system with sophisticated hardware, in the best case requiring only a different component configuration.

Design

CBA's design goals were most easily met with a recursive container-based design. A container provides an execution context for its contained objects. Contained objects model concepts from a problem domain, and may themselves be containers. This simple idea allows a tree of objects to be defined to act as an executable model. We refer to this repeating containment pattern and our approach to handling it as "Fractal Containment®. This concept is discussed in more detail in ZedaSoft's "Fractal Containment® Technical Brief" white paper.

At this point we started thinking about the types of contained objects that might be of use in a typical simulation:

Environment – a type of contained object that models the virtual world within which entities operate. The environment is independent of the entities participating in the simulation. For the simulation problem domain the environment may include a description of the earth model in use, the characteristics of the atmosphere, the layout of the terrain, a model of the sea state, and anything else an entity may need to know to perform its updates. The environment is a dynamic object that may change over time. For example, the characteristics of the atmosphere may change as a function of time of day, or the terrain description may change due to a munition detonation. Environment objects:

- Live as long as the container and cannot be destroyed or removed.
- Typically do not exist at only one point in space and do not move, though they may regularly update their properties.
- Do not regularly publish state data over an external data bus, and instead act as services that provide data to entities upon request.
- May provide network notification of events. For example, a terrain description may provide a notification that it has been modified due to a munition detonation.

Entity – a type of contained object that serves as a superclass for models of real-world things such as aircraft, sea vessels, and land vehicles. Entity objects:

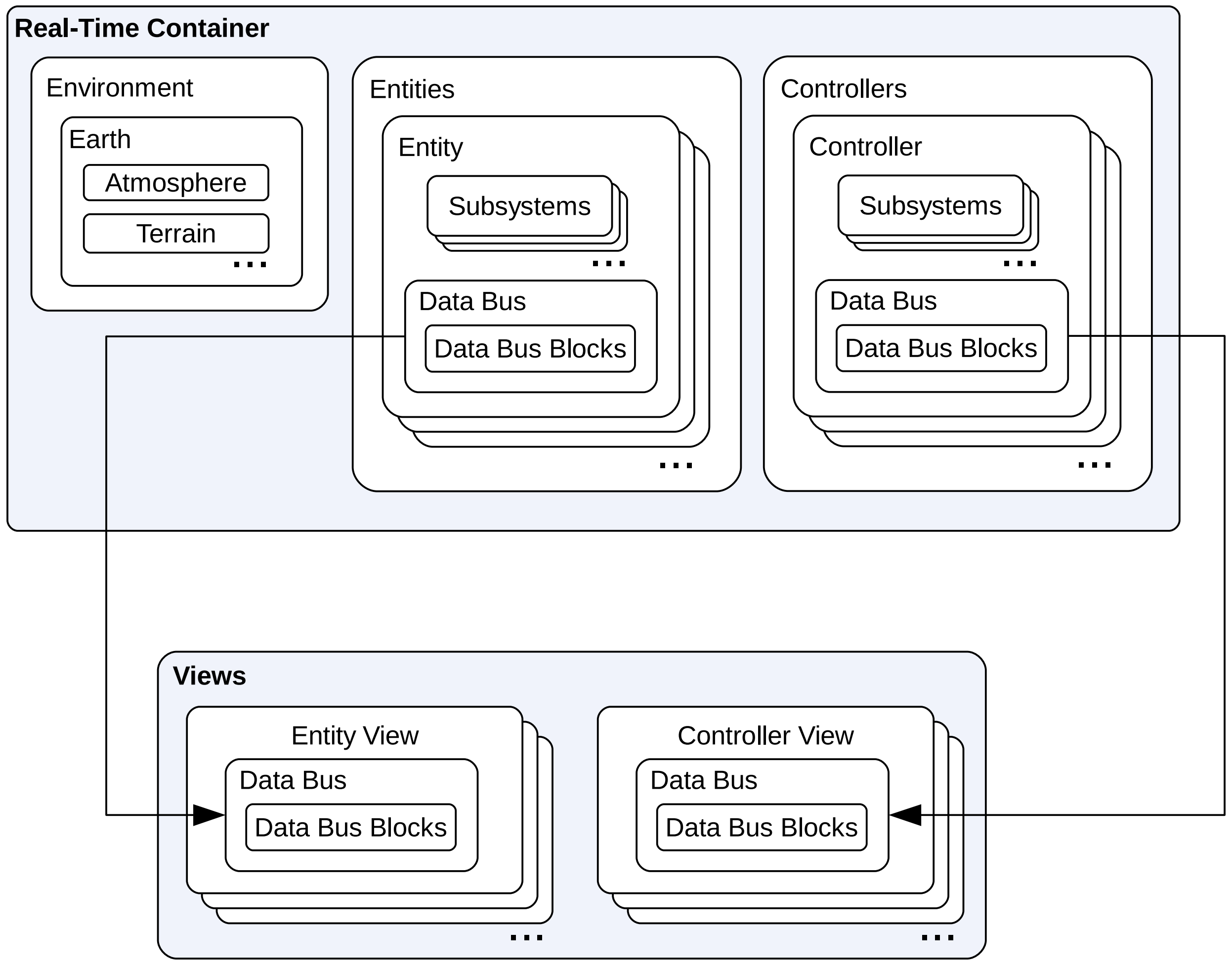
- Come and go during a container's lifetime. For example, when a missile is fired it becomes an autonomous entity in the container.
- Exist at a point in space and move about in the virtual environment.
- Maintain state information that can be regularly published on an external data bus.

Controller – a type of contained object that is not an entity but is still a managed object in the simulation. These objects operate in the container just like entities do, but don't represent real-world things. Examples include controllers for external devices such as image and audio generators, and scripting engines for scenario control.

View – a way to interact with and/or examine a remote contained object. Real-time containers and their contained objects do not provide any views. This job is left up to remote view applications. Views such as an operator station may provide a look at the interfaces of a single entity, while views such as a mission overview may provide a look at all entities.

Data Bus – a network-based memory reflection mechanism through which logical intra-entity data can be shared. Container-side data sources communicate with their views over a data bus.

An arrangement of these components is illustrated notionally below. The boxes suggest levels of containment in the object tree:



Containers

Containers provides services to their contained objects. These include, among other things:

Life-Cycle Management – each container manages the execution of its contained objects through a consistent life-cycle interface. This interface defines methods for:

- Notifying an object that it's about to be added or removed from a container;
- Notifying an object that other objects are about to be added or removed from a container;
- Telling an object when to perform its routine updates;
- Notifying an object that the container is about to terminate.

Additionally, the top-level container controls the main run loop and acts as an "executive" for the simulation.

List-Based Processing – containers maintain their contained objects in an open-ended list, making the processing power of the simulation host and the capacity of the network the only barriers to scale.

Categorization – a container can maintain subsets of its main contained object list based on categorization criteria such as class membership or property value evaluation. These categories are updated as objects come and go in the container. For example, a simple radar system may only care about entities that happen to be air vehicles. A category can be set up to maintain a subset of the main contained object list that meets this criteria. If another entity comes along later and asks for a category with the same criteria, then it will be given access to the same category. This prevents each object from having to maintain possibly duplicated subset lists.

Search Facilities – contained objects have access to the entire containment tree. Any container in the tree can be searched comprehensively using either predefined or user-defined search criteria. This permits objects to locate other objects without needing to define connections during compilation.

Publish/Subscribe Events – a container maintains a publish/subscribe event center where an arbitrary number of objects can register to receive notifications when events of interest take place. This permits the originator of the event to not have to care about which potential recipients are interested, and leaves it up to the recipient to decide how the event is handled. An example is munition detonation. When a munition detonates it posts a notification to the event center. The event center then notifies interested parties, who make a decision about how the event affects them. An image generation controller may take the opportunity to create an explosion in the visual scene. An audio controller may create an audible explosion. Objects that represent real-world objects may perform damage assessment. This mechanism contributes to the extensibility of the system by keeping the object that posts the notification from having to know what other objects need to receive it. If new types of objects that care about a particular type of notification are added to the system, then they simply register to be notified and nothing else has to change.

Execution Sandboxes – object execution is performed in a sandbox that protects other objects from error conditions. If an object encounters a fatal error, then it's simply removed from the container.

Views

The real-time container is a data generation engine and doesn't perform any graphics tasks. The job of presenting data to the user is left up to views that know how to interface with peer objects in the real-time container and accept data from them over the network.

This is an important concept in CBA because it allows real-time data generation to be partitioned away to separate (possibly headless, embedded, or otherwise specialized) machines, and makes possible a “one real-time container – many views” model.

A view may show information about the entire container, a single object within the container, or anything in between. The initial design of CBA included three main view applications:

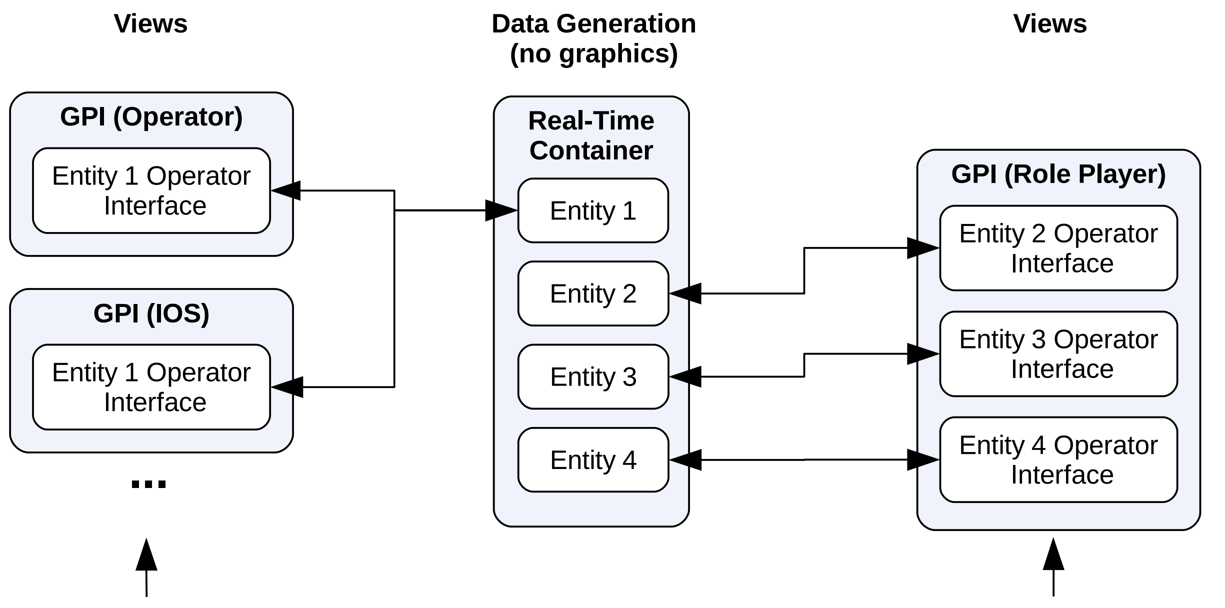
- Graphical Participant Interface (GPI) – a container for the display set of a single entity. Each entity plug-in can provide an optional display set that defines the graphical presentation of its operator interface views. For example, an aircraft entity may provide views of the various cockpit displays with which the pilot can interact, including virtual bezels and panels. When the GPI starts it discovers running simulations via a discovery service, and allows the user to select an entity of interest from a particular simulation. The entity's operator control display set is then dynamically located and brought to life on the GPI user's station.

- Operator Console – a multi-entity view that displays a list of all objects in the simulation whether or not they represent real-world objects. Plug-ins can provide editing and/or inspection views that can be accessed from the Operator Console application. The user can also monitor and control the real-time container's state from this view.

- Mission Overview – a multi-entity view that provides a simple God's-eye view of all objects in the simulation that represent real-world objects. This view is conceptually similar to the Operator Console view, except a subset of the objects in the remote simulation is shown, and a graphical presentation is used instead of a list.

A future version of CBA will combine these views together into a single unified viewer.

The only restrictions on the number of views that can interact simultaneously with a given container are available network bandwidth, and the ability of the objects in the container to prepare, send, and receive data. Multiple views can be connected simultaneously to the same or different entities and can come and go at the operator's discretion. This provides support for features such as manned role player stations that allow the operator to control multiple entities, or display repeaters to support an Instructor/Operator Station (IOS).


Views
Data Generation (no graphics)
Views
GPI (Operator)
Entity 1 Operator Interface
GPI (IOS)
Entity 1 Operator Interface
...
Real-Time Container
Entity 1
Entity 2
Entity 3
Entity 4
GPI (Role Player)
Entity 2 Operator Interface
Entity 3 Operator Interface
Entity 4 Operator Interface

Multiple views can connect to an entity simultaneously.

The limitation on the number of connections is network bandwidth and processing power.

A single station can connect to multiple entities simultaneously.

In legacy simulations these capabilities often required specialized hardware such as video distribution systems. In CBA we handle it with software component distribution across the network.

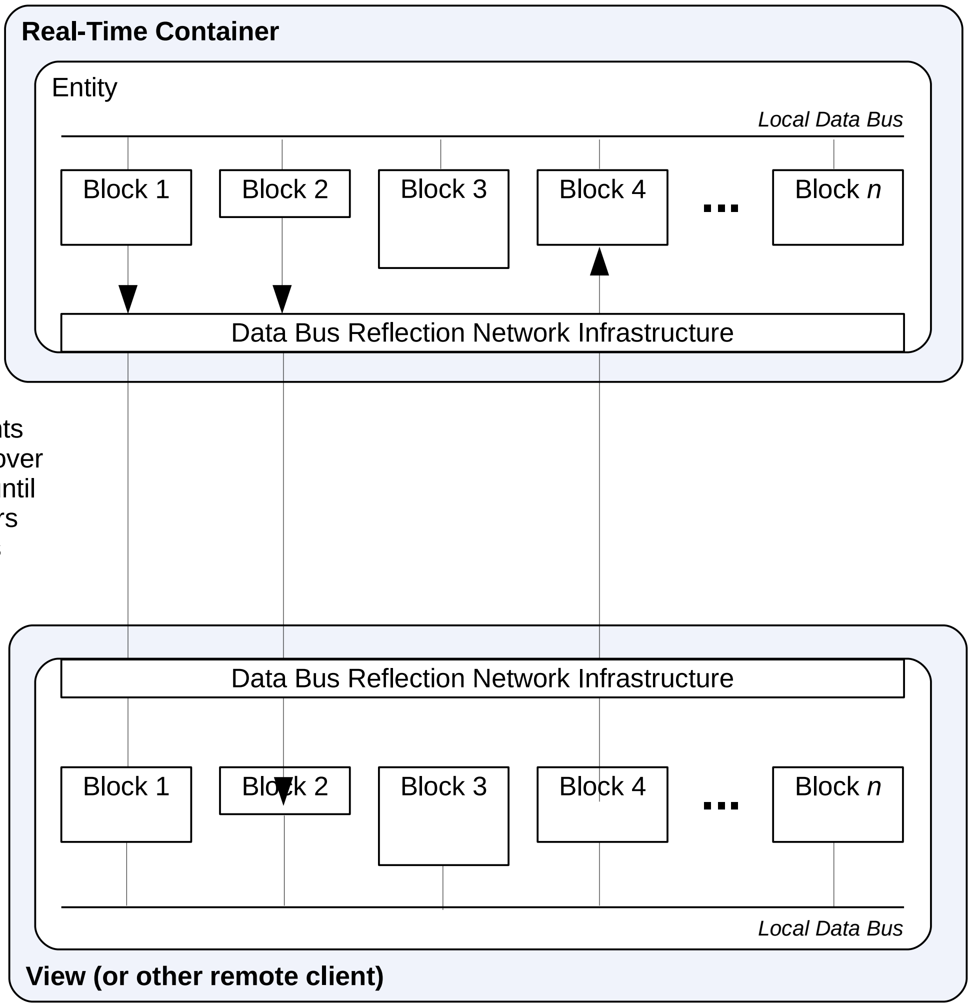
Data Bus

By default, CBA view components communicate with their peer data sources in the real-time container via what is essentially a network-based reflective memory system that we refer to as "data bus". When a remote view starts, it negotiates a communication channel over which data can be sent. This channel is used to support a distributed publish-subscribe system where data blocks on the bus can be requested by remote views and other client applications. From a view's perspective the data exists locally as if the view were a part of its peer in the remote real-time container. This mechanism is what allows views and their container-side peers to be distributed across the network.

The entity model in the real-time container "owns" the data bus. Without it there is nothing to which remote clients can connect.

Only blocks to which remote clients have subscribed are transferred over the network. Transfer continues until all subscribers vanish. Subscribers are tracked per data block. In this example the remote view has subscribed to blocks 1 and 2.

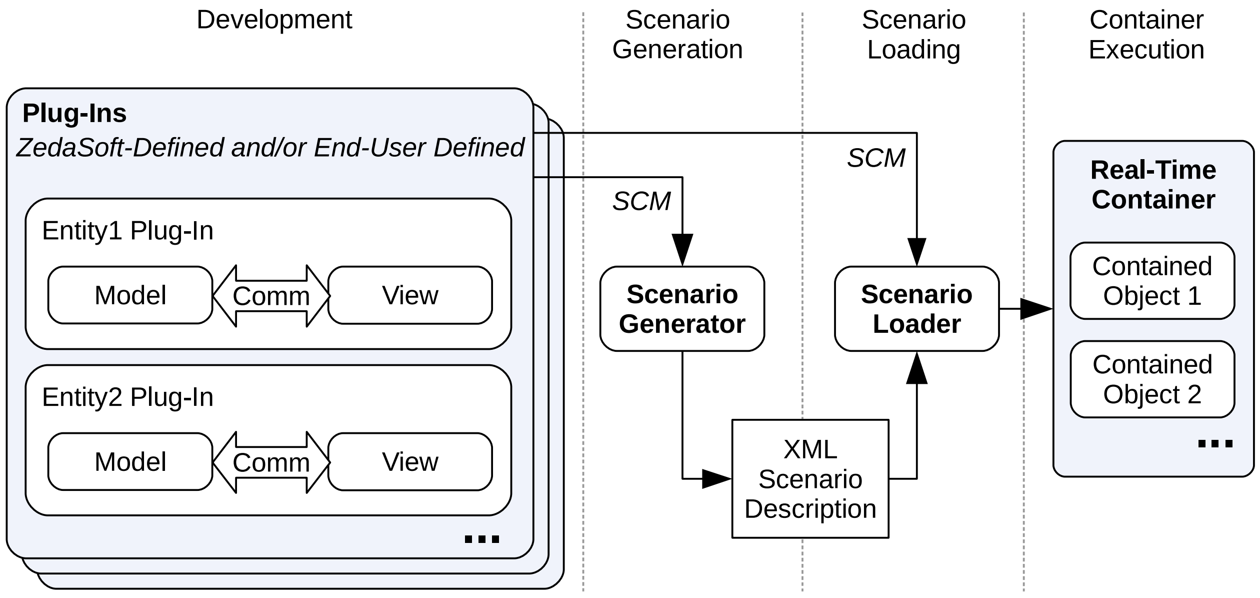
Blocks a remote client publishes (Block 4 in this example) are always transmitted to the connected entity. The entity doesn't have to subscribe explicitly.



Data bus is the default built-in communication mechanism in CBA, but it is just one possible communication channel. Views and their container-side peers can use any communication technique they want as long as it is understood by both endpoints and is well-behaved.

Real-Time Container Creation and Execution

CBA provides a mechanism named Software Component Manager (SCM) through which plug-ins can be located and loaded from the local filesystem or network. The information provided by the plug-ins is used to create scenario descriptions and build executable real-time containers.



The Scenario Generator allows users to interact with entity plug-in information to create scenario descriptions. Scenario descriptions specify what instances need to be created to build a container and what values should be used to initialize those instances. Scenario descriptions are stored in plain-text XML files and assembled by a mechanism we refer to as “BeanMachine”. BeanMachine is discussed in more detail in ZedaSoft's "BeanMachine Technical Brief" white paper.

The Scenario Loader is responsible for creation and execution of simulation containers. The loader consumes an XML scenario description file, locates the required classes either locally or over the network, creates instances as specified in the scenario description, and through a run-time interrogation mechanism determines information about the involved classes and establishes any specified initial values. The loader then starts the container’s main execution loop, at which point remote clients such as views are free to connect to the container and/or its contained objects and interact with them.

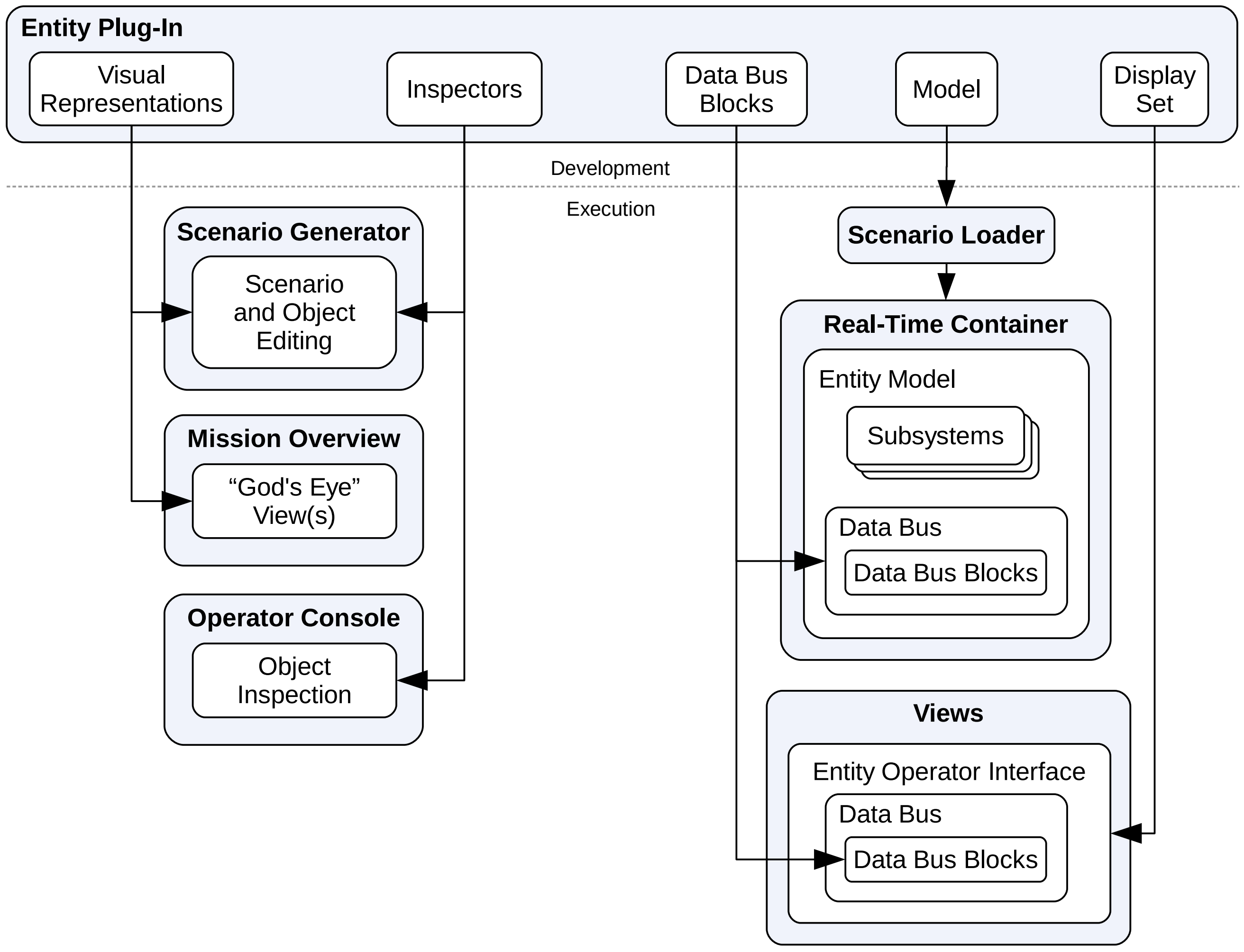
The Scenario Loader is capable of interacting with any classes that follow a few simple implementation guidelines, even if those classes were unknown to the loader prior to encountering a reference to them in the scenario description.

Container Extensibility

New types of objects are added to the container by implementing a plug-in. The major components of the plug-in are the real-time components that implement the object's model, the view components that implement the object's inspectors and operator interfaces (if any), and the data bus components that are used to transfer data between them. Note that the communication channel between the real-time components and view components doesn't have to use data bus; it can be anything as long as the endpoints agree on the mechanism and it's well-behaved.

Both the container-side and view-side components are defined in terms of interfaces, so the real-time container and view containers can interact with any new component as long as the interfaces are implemented correctly. This allows third-party users to add classes to the system the same way we do it at ZedaSoft.

The following illustration shows a plug-in and how its constituent components are distributed for execution:



A key goal of using the plug-in approach was to solve another problem that existed in legacy simulations we had worked with. In those systems it was common for the introduction of a new type of object to cause a cascade of changes across the code base. For example, introduction of a new type of entity may cause a mission overview to need a new kind of graphical representation for that entity, and cause a scenario generation system to need a new set of inspection panels for that entity's attributes. We thought there was something fundamentally wrong with needing to disturb stable, fielded applications just to expand the kinds of things that can participate in the simulation. The plug-in approach solved this problem by defining the interfaces through which applications could interact with objects, and defining applications that needed to interact with those objects in terms of these generalized interfaces. This allowed new types of objects to be added without disturbing the applications, as long as the objects' defining classes implemented the interfaces properly.

In CBA there is no inherent limit to the number of plug-ins that the system can support simultaneously; the only practical limit is the available resources of the simulation host. Furthermore, plug-ins can be loaded from a variety of locations. For example, plug-ins can be loaded from the local file system, from another machine on the local network, or even over the web provided accessible network paths are provided.

A running container can load an accessible plug-in at any time. This means that objects of any kind can be brought to life in a container both at start up and while the container is running.

Summary

The design choices we made for CBA have exceeded our expectations in practice, and have been a significant contributor to ZedaSoft's ability to deliver simulation products on tight timelines.

CBA’s ability to assemble arbitrary component sets together allows the core system to be extended without modification. The result is an extensible development and execution framework that can be applied to many simulation problem domains. CBA’s network-centric approach to data sharing, peer approach to entity modeling, and list-based processing provide levels of flexibility that traditional simulation designs can’t support.