A unified component framework for dynamically extensible virtual environments

Kapolka, Andrzej

A Unified Component Framework for Dynamically Extensible Virtual Environments

Andrzej Kapolka, Don McGregor, Michael Capps

The MOVES Institute
Naval Postgraduate School
Monterey, CA 93943, USA
+1 831 656 {2253, 4090, 2865}

{akapolk, mcgredo, mcapps}@nps.navy.mil

ABSTRACT

If large-scale shared virtual worlds are to be established on the Internet, they must be based on technologies that allow them to adapt, scale, and evolve continuously—that is, without their being taken offline. In the course of designing NPSNET-V, an architecture intended to satisfy these criteria through component-based dynamic extensibility, the authors recognized the need for a consistent, unified component framework. This framework, which they implemented in Java™, allows one to construct applications as component hierarchies rooted at an invariant microkernel. A simple extensible interface layer and event model allow components to communicate with one another, and an XML configuration and serialization mechanism permits applications to store and transmit component and application state in a versatile standardized format. After an initial bootstrapping process, one may add, remove, and upgrade components at run time, and one may introduce newly loaded Java™ code anywhere in the application hierarchy at any time. The complications posed by this reconfigurability and the hierarchical nature of NPSNET-V applications led the authors to develop a consistent design strategy, which they based largely on several common design patterns. The most critical design pattern that they used was the *Model-View-Controller* pattern, which forms the basis of the NPSNET-V entity model.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*frameworks, patterns*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*virtual reality*

General Terms

Design, Languages

Keywords

Component-based architectures, dynamic extensibility, networked virtual environments, Java, XML

CVE '02, September 30–October 2, 2002, Bonn, Germany.
ACM 1-58113-489-4/02/0009

1. INTRODUCTION

1.1 Motivation

The software required to create a networked virtual world may consist of many different elements—client applications, server applications, and middleware libraries, to name a few—but all share a single limitation: unless their authors have explicitly designed them with extension in mind, their feature sets are completely fixed at the time they are compiled and delivered to the end user. Many applications have no need for the ability to be extended after deployment. Users may upgrade small client applications, for instance, simply by replacing them with a newer version. Larger applications may provide patching utilities that take the application offline and replace only certain portions of the binary executable. However, an increasing number of modern applications provide support for run-time extension in the form of plug-ins: downloadable program components that extend or enhance the functionality of their host applications. Typically, applications that support plug-ins provide only a small number of "hooks," or application areas in which plug-in functionality may be incorporated.

In contrast, virtual environment applications offer an almost limitless number of opportunities for the inclusion of plug-in technology. Graphical plug-ins may generate 3D models on the fly procedurally, competing with declarative geometry description languages; networking plug-ins may provide support for new protocols and filtering schemes; plug-ins for physical simulation may introduce previously unknown forces; plug-ins for artificial intelligence, when applied to the simulated inhabitants of the virtual world, may provide the user with unexpected new challenges. By layering and aggregating plug-ins, it should be possible to create software in which the only functionality not provided as a plug-in—that is, the only section of code that one cannot replace at run time—is a minimal binding mechanism, or microkernel.

The most critical role of plug-ins in virtual environment technology may be their use in the creation of Internet-based persistent shared virtual worlds. Currently, the content and scale of such worlds are limited by the standards or software upon which they are based. A world based on the Distributed Interactive Simulation (DIS) standard, for instance, may only contain the entities defined within that standard. In order to introduce a new type of entity, the world must be brought offline, the standard revised, and the supporting software updated and redistributed. A world supported by fully extensible software, on the other hand, can remain online indefinitely. As its technical requirements and the needs of its users change, and as its developers upgrade and extend its functionality, the world will maintain the continuity of its existence and preserve its essential state despite the evolution of its constituents.

1.2 Background

NPSNET-V [2], an architecture intended to provide a supporting platform for such extensible virtual worlds, had been in development for more than a year before its authors noted an important pattern: one by one, they were converting critical system components from statically linked classes to dynamically loadable modules. At first they intended only virtual entities—such as tanks, planes, and human avatars—and network protocols to be loaded at run time. Later they added support for the dynamic definition of area-of-interest management strategies. Unfortunately, because they lacked a consistent framework for component-based development, they were forced to "componentize" separately each desired type of module, adding time and complexity overhead to their development efforts. After several iterations of this reinvention process, they realized that they would benefit significantly from creating a unified, all-encompassing component framework: a system in which every application could be implemented as a federation of dynamically loadable modules, loosely coupled by a minimal set of well-defined relationships. In other words, faced with the daunting task of independently redesigning elements of NPSNET-V as pluggable components, they chose to reinvent the architecture as one in which all components—save for a minimal microkernel—inherit from a common base class the ability to be loaded and composed at run time.

1.3 Requirements

Although many types of applications might benefit from being based on such a system, the authors of the NPSNET-V component framework chose to focus their efforts on satisfying the specific requirements of networked virtual environments and, in particular, NPSNET-V: a platform for student development and a test bed for advanced virtual environment research. Any middleware intended for student use must be as simple and consistent as possible, allowing its users to achieve competency within the duration of an academic quarter. Any research environment must be versatile and reconfigurable, allowing researchers to experiment with many different technologies and strategies without having to modify the environment's underlying framework. That framework must build upon preexisting standards in order to maximize its interoperability with other systems and to minimize the amount of effort spent by its developers in recreating functionality available elsewhere.

Additionally, virtual environment architectures—particularly those intended to support large shared virtual worlds over long periods of time—share a number of challenging technical requirements. Because virtual environment applications demand high performance, they require an underlying architecture that is efficient and that encourages efficient extension. Because shared virtual worlds vary widely in size and number of participants, the architecture must be scalable. The architecture must provide a persistence mechanism so that world state is not lost when no participants are present within the world. It must be version-safe, because large and long-lived virtual worlds tend to incorporate different versions of the same components. It must encourage composability, so that one may easily and effectively combine worlds and world components developed by different organizations. Finally, and perhaps most importantly, it must be dynamically extensible; that is, to as large an extent as possible, the architecture must permit the seamless run time extension and replacement of any part of its hosted application.

1.4 Previous Work

Several existing architectures have addressed this dynamic extensibility requirement. Bamboo [9], a multi-platform, multi-language plug-in framework, was the first system to provide a microkernel-based virtual environment architecture in which all system elements aside from the kernel were pluggable components that one could add, remove, and replace at run time. To maximize efficiency, Bamboo's components run as native shared libraries, distributed as portable source code and compiled at run time by a sophisticated language loading mechanism. Unfortunately, this very flexible approach incurs a great deal of complexity, particularly in terms of facilitating communication between components written in different languages. The most effective means of performing such communication is the Common Object Request Broker Architecture (CORBA) [10], in which component interfaces are described in a separate, common interface description language that may be used to generate bindings for as many different languages as are used by the application's components. CORBA offers a number of useful services, and supports location transparency: the ability to interact with components over the network in the same manner as one would interact with local components. However, CORBA is a heavyweight system that developers often find difficult to learn and employ.

Bamboo's components make extensive use of callbacks in communicating with one another—a technique that is also fundamental to the GNU/MAVERIK architecture [4]. Like Bamboo, MAVERIK is a microkernel-based system; however, the extent to which MAVERIK supports dynamic extension is unclear. The Deva system [6] extends MAVERIK with functionality intended to support networked virtual environments. Both MAVERIK and Deva feature a number of innovations applicable to component-based virtual environment applications hosted within any framework: MAVERIK, for instance, features a powerful immediate-mode rendering model in which components implement different culling methods and display algorithms, and Deva offers a unique behavior model that differentiates between each entity's innate behavior and the behavior that it acquires from its environment. For simplicity and portability, the MAVERIK kernel consists of C code only; Deva is written in C++.

The designers of the Java™ Adaptive Dynamic Environment (JADE) [5] chose to base their architecture entirely on Java™: a platform that allows code to be compiled to an intermediate language and executed on any system capable of running a Java™ Virtual Machine. JADE also provides a number of abstractions useful for any component-based architecture. For instance, each JADE component is subject to a well-defined lifecycle and exists within a unified containment hierarchy. This principle of containment is also fundamental to the Extensible Runtime Containment and Services Protocol (ERCSP) [7], an extension to the JavaBeans™ component standard [8]. The ERCSP, however, augments its containment model with provisions for the registration and location of *services*, defined as Java™ interfaces and implemented by dynamically loadable service providers. The authors of the JavaBeans™ standard have extended it recently to include an XML serialization mechanism: a crucial feature, as traditional Java™ serialization lacks the version-safety required for long-term component storage and the transmission of component state between heterogeneous clients.

Still, although the JavaBeans™ component model, like the other systems discussed above, provides a number of useful and important features, the authors felt that no one existing architecture could satisfy all of NPSNET-V's requirements independently. They decided, therefore, to build an entirely new framework, incorporating elements from all of the existing architectures that they surveyed. For example, it is often difficult to develop complex applications using JavaBeans™, particularly when extensive component-to-component communication must be performed, because the JavaBeans™ property model is optimized for human editing. Beans are intended to be largely independent, and although the ERCSP addresses this issue to an extent by providing a contextual framework in which to place beans, the ERCSP itself is complex and unwieldy. However, the authors adapted the ERCSP service model, along with the support for XML serialization and simple property editing offered by the JavaBeans™ standard, for use in the NPSNET-V component framework. Similarly, while the authors felt that the Bamboo architecture lacked the simplicity and consistency necessary for NPSNET-V, they realized the usefulness of basing their framework upon an invariant microkernel and providing all additional functionality in the form of pluggable components. JADE provided the principal foundation for the NPSNET-V component framework, although its lack of a persistence mechanism prevented the authors from using it directly. The NPSNET-V component framework owes much to JADE, including its concepts of hierarchical containment and the component lifecycle, and much of its terminology.

2. OVERVIEW

Like JADE, the NPSNET-V component framework is based entirely upon the Java™ platform and hosts applications as component hierarchies, or *application graphs*, rooted at an invariant microkernel. As in JADE, the atom of composition and extension is the `Module`: an abstract base class, included with the kernel, that all components must extend. The `ModuleContainer`, a subclass of `Module` also included with the kernel, is not abstract; developers may either use it as a generic container or extend it to create specialized containers. The `Kernel`—a singleton module, derived from `ModuleContainer`, that acts as the root of the containment hierarchy—is home to the framework's only `main` method, the single point of entry that initializes all applications through a bootstrapping process described in a later section of this paper.

Any application constructed using the NPSNET-V component framework thus consists of a federation of loosely coupled modules arranged in a strict containment hierarchy with a more than superficial resemblance to a file system directory structure. The `Kernel` acts as the root directory, and all other `ModuleContainer` instances act as subdirectories. Modules may be added to or deleted from containers at any time, and removal operations are recursive; that is, removing a module container also removes its contained modules. All modules are required to possess instance names, and each module in the containment hierarchy may be addressed using absolute or relative paths conforming to typical Unix path conventions: for instance, the absolute path `/modelCore/entityModel` addresses a module named `entityModel` that is a child of a container named `modelCore`, and the relative path `../viewCore` addresses a module named `viewCore` that is a child of the resolving module's container.

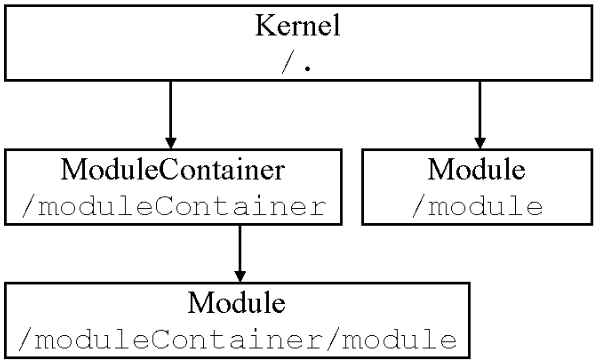


Figure 1. Module containment and naming hierarchy. Boxes represent modules within the framework; arrows indicate containment relationships. Each module has both a class name (above) and an instance path (below).

When a module resolves a relative path, it does so through its parent in the containment hierarchy, which represents the contained module's context within the application. By convention, modules are grouped in containers according to their functional roles, and are expected to maintain awareness of their neighboring modules in order to establish implicit relationships with them. Modules within the same context have easy access to each other and share resources, including a common system-level interface: their container. Containers are responsible for identifying modules by their names and properties, locating services for their contained modules, and controlling their contained modules' lifecycles.

Each module is subject to an explicitly controlled lifecycle modeled closely after that used in JADE and the Java™ applet model. All NPSNET-V modules must have no-argument constructors, and in addition to the construction/finalization operations common to all Java™ objects, NPSNET-V modules must be explicitly initialized and destroyed using their `init` and `destroy` methods. Modules that operate threads of execution—as well as all container modules—implement two additional methods, `start` and `stop`. In the case of container modules, these methods act recursively, starting or stopping entire application subgraphs. Finally, in order to support seamless module upgrading, modules implement the `replace` and `retire` methods. When a container must replace an old module with a newer version, it calls the `replace` method of the new module instead of `init`, and the `retire` method of the old module instead of `destroy`, allowing the two modules to perform a cooperative hand off operation. Once the new module has absorbed the state of the old module and has notified all dependent modules of the succession, it may interact with the other components of the application as if it were no different from the replaced module.

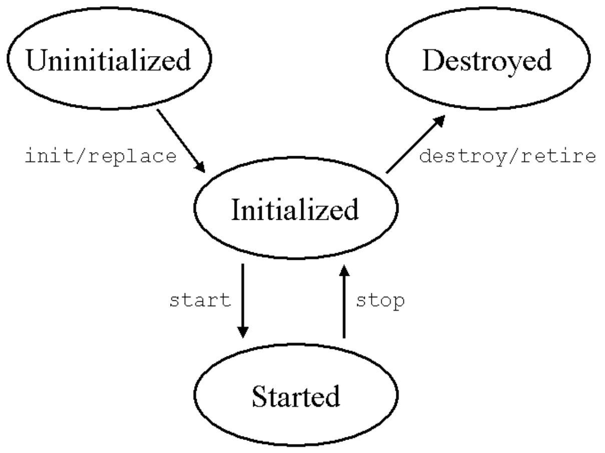


Figure 2. Module lifecycle. Ellipses represent lifecycle states. Arrows indicate transitions between those states, and arrow labels indicate module methods associated with the transitions.

3. INTERFACE LAYER/EVENT MODEL

In order for dynamically loaded components to interact, however, they must agree to use a common interface layer. This interface layer consists of a number of invariant Java™ interfaces that may be implemented by modules wishing to provide access to their state and control mechanisms. For instance, the `Startable` interface defines the `start`, `stop`, and `isRunning` methods to control and query the activation state of a threaded module. A basic interface collection is included with the kernel, but other interfaces may be added to the system at run time. Unlike modules, interfaces are versionless; once an interface has been defined and published, it cannot be changed. This means that new interfaces, when encountered, may be downloaded and permanently added to the framework without fear of their being removed or redefined.

The most fundamental type of shared interface used in NPSNET-V is the *property*. Each property, defined as a subinterface of `PropertyBearer`, represents a specific aspect or quality of its implementing modules. The `Transformable` property, for example, defines two methods: `getTransform` and `setTransform`, each of which takes a `Transform3D` (defined in the Java 3D™ libraries) as an argument. Properties may be used to locate modules of interest; for instance, any module may easily retrieve from its container a list of all modules within that container that bear a specified property. As compared to the JavaBeans™ property model, in which properties are defined by name and accessed by simple `get` and `set` methods, defining properties by interface is a more versatile approach that allows easier communication between modules. However, for lightweight properties, NPSNET-V supports the JavaBeans™ property model as well. Modules may implement the `BasicPropertyBearer` interface, which defines a single method: `getBasicPropertyNames`. Each name returned is taken to be a lightweight property that may be read and written using its corresponding `get` and `set` methods.

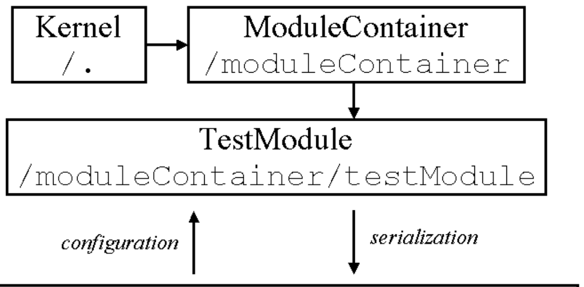
Among the properties that modules may implement are special properties, *services*, that announce the module's ability to perform a critical application role. Each service is defined as a subinterface of `ServiceProvider`. The `TimeProvider` service, for instance, consists of an interface that defines a single method, `getCurrentTime`. The principal difference between properties and services is that, while many modules within a single context may implement the same properties, only one module can provide any given service. Also, containers inherit service providers from their supercontainers. If a module requests a service provider from its container, that container first searches within its own context, then, if no service provider is found, delegates the request to its parent in the containment hierarchy. In this way, general-purpose service providers defined near the root of the application may be overridden by specialized service providers defined further down the hierarchy. As an example, a default `TimeProvider` loaded directly under the `Kernel` would provide the current time as known to the local system, but a specialized `TimeProvider`, its influence limited in scope to modules participating in a networked simulation, would provide a network-synchronized time reading.

When services are registered and deregistered, and when module properties are modified, the system must somehow notify all parties that may be affected by the change. NPSNET-V accomplishes this by providing a synchronous event model that relies heavily on Java™'s powerful reflection capabilities. Each module maintains a list of event listeners for every type of event that it generates. For example, instances of `ModuleContainer` generate events when service providers are registered or removed. To receive these events, listeners may subscribe to all events from that container, to events related to module containment (subclasses of `ModuleContainerEvent`), or to one specific type of event, `ServiceProviderChangeEvent`. Modules fire the `PropertyChangeEvent` when their properties change, and listeners concerned with specific properties may register to receive only events associated with their properties of interest.

4. CONFIGURATION/SERIALIZATION

The interface layer and event model facilitate component-to-component interaction, but they do not define a method by which component relationships may be defined and stored; for this, the NPSNET-V component framework integrates an XML-based configuration and serialization mechanism. Through this mechanism, the state of any part of a running application may be serialized and stored as an XML file, which may be loaded at a later time, or on another client, to recreate the stored configuration. XML offers at least three major advantages over proprietary binary serialization formats: its textual nature allows for easy human editing and debugging, its extensibility ensures its adaptability and version-safety, and its popularity as a standard has encouraged a proliferation of freely available XML tools. The XML configuration files read and generated by NPSNET-V are used for initial application bootstrapping, and may be used for persistent storage of world components, for transferring component state between applications over the network, and for other related tasks.

The `applyConfiguration` and `getConfiguration` methods of class `Module` provide the means to activate the configuration and serialization mechanism. A module may provide specialized handling by overriding these methods; for instance, the ModuleContainer class recursively serializes its contained modules within its implementation of getConfiguration, so that any part of the application graph may be serialized by calling the getConfiguration method of its root container. By default, configuration and serialization are largely handled by the extensible ConfigurationElementInterpretationProvider and PropertySerializationProvider services. These services manage handlers that interpret XML elements according to their tag names and serialize modules' state according to the property interfaces that they implement. For instance, the PropertySerializationProvider encodes any Transformable module's transform upon serialization by calling the module's getTransform method and converting the result to an XML <Transform/> element. Upon loading a configuration that contains such an element, the ConfigurationElementInterpretationProvider decodes the element and calls the target module's setTransform method.



```
<Configuration>
  <Body>
    <Container name="moduleContainer">
        <Module class="TestModule" name="testModule"/>
    </Container>
  </Body>
</Configuration>
```

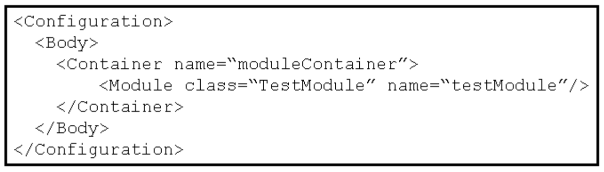


Figure 3. Configuration and serialization. The upper half of the figure represents a running application, while the lower half contains the kernel's serialized configuration.

As an example of the usefulness of combining this configuration and serialization mechanism with a hierarchical component framework, NPSNET-V includes the ConfigurationServer module: an HTTP server that provides access to the configuration of running NPSNET-V applications. Once the ConfigurationServer is loaded, the user may retrieve a serialized representation of any part of the application graph using a URL, such as http://hostname/modelCore/entityModel, that addresses an application component. In addition to the HTTP GET and HEAD methods [1], the server supports the POST method, allowing one to apply, as well as generate, application configurations. Combined with a web browser, the ConfigurationServer acts as an effective debugging tool; however, its primary utility lies in its ability to transfer component and application state between clients. For instance, one may replicate the entire state of a running application by starting NPSNET-V with the URL of a remote client on the command line.

5. BOOTSTRAPPING/EXTENSION

The NPSNET-V component framework initializes itself through a bootstrapping process driven by one or more application configurations. The invoking user must specify each configuration on the command line, either as the path of a local file or as the URL of a web resource. The kernel applies each configuration to itself in sequence upon startup. Typically, each configuration includes several <Module/> elements, each of which causes the kernel to load and initialize—and, when necessary, start—a new module. If the newly loaded module is a container, it may itself load modules. Configuration files indicate this behavior using nested <Module/> tags.

The first module loaded must usually be a ResourceLocationProvider: a service provider module that locates and loads resources, including module classes, according to their names and version numbers. The <Module/> element that causes the kernel to load the ResourceLocationProvider must include a package attribute specifying the URL from which the module class may be retrieved. All subsequent <Module/> elements, however, need only specify the name of the desired module class; the ResourceLocationProvider provides the corresponding URLs. Each resource is uniquely identified by its fully qualified name (such as org/npsnet/views/animals/Shark.wrl) and its version string, which must conform to the standardized format used in Java™ manifest files. Version strings may be compared for recentness, allowing the ResourceLocationProvider to locate the most recent versions of requested resources. Resources must be bundled in jar (Java™ archive) files that include manifests describing their content. The NPSNET-V manifest format builds off of that used in the Java™ platform, adding support for the representation of module capabilities and dependencies.

In addition to the ResourceLocationProvider, the kernel must usually load several other modules providing system level functionality before the component framework can achieve robustness. For instance, the kernel implements the ConfigurationElementInterpretationProvider service, but it is only able to interpret four XML tags upon startup: <Module/>, <Container/>, <Include/>, and <Debug/>. In order to process other tags, the kernel must load additional modules that extend the configuration element interpretation service. Similarly, the kernel cannot serialize module properties upon startup; for this, it must load an implementation of the PropertySerializationProvider service. The related PropertyTransferProvider service provides a means by which property values may be automatically transferred between modules during the upgrade process—a process that may be controlled using an implementation of the ModuleUpgradeProvider service.

Typically, a ModuleUpgradeProvider periodically checks all loaded modules for more recent versions, upgrading each module when necessary. One may also manually activate the upgrade process using, for example, a console command. When a module must be upgraded, the upgrade provider creates an instance of the new module version and orders the module's container to replace the old instance with the new. In doing so, the container coordinates a hand off procedure in which the new module assumes the state and application role of the old module, typically using the PropertyTransferProvider to copy state information according to the properties shared by both modules. To notify dependent modules of the transfer, the old module fires a ModuleReplacementEvent before retiring.

In addition to being able to activate the upgrade process, any module may extend the system at any time by loading new modules and attaching them to the framework. Modules may do this either indirectly, by applying configuration documents to module containers, or directly, by instantiating and registering the modules themselves. Each loaded module may operate threads of execution, provided that it implements the Startable property interface. When all threads of execution have ceased, or whenever a module requests that the application be terminated, the kernel disassembles the framework, recursively stopping and destroying all loaded modules before exiting the application.

6. DESIGN STRATEGIES

Applications constructed using the NPSNET-V component framework may assume a variety of different forms, but in order to manage the complexity incurred by their reconfigurable nature, they must base their configurations and their components on a set of consistent design strategies. Many of the strategies employed by NPSNET-V components are based on *design patterns* [3]: well-understood, reusable elements of high-level software design. Other strategies take advantage of NPSNET-V's containment hierarchy to simplify relationships between components. For instance, whenever possible to do so without sacrificing versatility, components allow one to define relationships implicitly through their containment configurations. When one loads a Channel (a module that represents a communications link) as the child of a NetworkController (a module that forms and transmits network updates), the two modules connect automatically, and the NetworkController begins to transmit and receive messages through the Channel. Similarly, one may add a GLView, a module used as an OpenGL scene element, to a rendering traversal by loading it as the child of a GLView that is already being traversed. In this manner, the containment hierarchy shapes the principal flows of both control and data within the application.

One may use modules conforming to the *filter* pattern to affect this flow. Within NPSNET-V, filter modules transfer control or data between their parents and their children, usually applying some transformation in doing so. For example, the AggregatingChannel, when placed between a NetworkController and another Channel module, collects the packets received from its parent and stores them in an aggregation buffer until that buffer reaches a certain size threshold, then transmits the entire contents of the buffer to its child Channel. When the AggregatingChannel receives aggregated incoming packets from that Channel, it splits them into their component subpackets and forwards them to its parent NetworkController.

The *façade* pattern is another pattern of which NPSNET-V components make extensive use. Façade modules act as fronts for the subgraphs that they contain, providing unified interfaces for collections of related modules. The SwitchingCamera, for instance, is a Camera that may contain any number of child Camera modules. The Camera interface describes several attributes, such as field of view, common to all cameras. The SwitchingCamera, when asked to return these attributes, returns the values corresponding to its currently selected child Camera. Modules may cycle between the SwitchingCamera's child cameras by invoking its previousCamera and nextCamera methods.

7. ENTITY MODEL

Cameras are examples of *entities*: modular constructs that conform to the *model-view-controller* pattern and that represent individual elements of virtual worlds hosted within the NPSNET-V architecture. The model-view-controller pattern requires that the abstract state of each entity, its model, be separate from its views, which depict the entity to the user, and from its controllers, which control the entity's state. Within NPSNET-V, each model must implement the EntityModel property interface. Views and controllers must implement the EntityView and EntityController interfaces, respectively. The view and controller interfaces extend Targeted: a generic property interface used by any module that relies on a target. In the case of views and controllers, that target is the model.

In addition to the model-view-controller pattern, two other patterns perform crucial roles in the NPSNET-V entity model: the *observer* pattern and the *remote proxy* pattern. The observer pattern, in conjunction with the NPSNET-V event model, allows models to notify their dependent views and controllers of changes to the model state by firing a PropertyChangedEvent. A view module that must update its visual depiction of the entity when the entity moves, for instance, may subscribe to be notified of all updates to the model's Transformable property. Likewise, a NetworkController may subscribe to its model's event stream in order to transmit the entity's state updates across the network. Entity state replication within NPSNET-V depends upon the remote proxy pattern, which allows a local replica model, or *ghost*, to act as a stand-in for a remotely owned *master* model. Modules, including views and controllers, may interact with the ghost as if it were controlled by the local client, but each state-modifying interaction that they attempt must be forwarded to the remote owning application, which maintains the definitive representation of the entity's state.

Entities, like modules, may form hierarchical containment structures. A world may be an entity, as may a vehicle within that world and an avatar within that vehicle. Typically, the model graph of an application is wholly or partially mirrored by its view and controller graphs. Each graph must be rooted at a core module—a model core, a view core, or a controller core—that is responsible for managing all modules beneath it. Views and controllers whose targets are containers listen for the registration of new modules within those containers in order to create corresponding children of their own. In order to create these children, container views and controllers require the models to bear *associated module configurations*: stored configurations that correspond to specific view and controller types. An entity model may store any number of associated module configurations, but may hold only one configuration for each type of view and controller: a GLView configuration for OpenGL rendering, for instance, a TextView for textual representation, and a

`KeyController` to allow users to manipulate the entity's state using the keyboard. One may add new associated module configurations to entity models at any time.

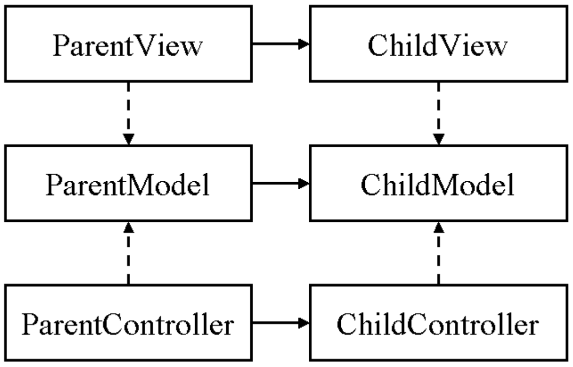


Figure 4. Entity model. Solid lines indicate containment relationships; dotted lines indicate targeting relationships.

Although these associated module configurations may be stored along with the entity model using the standard XML serialization procedure, a specialized mechanism exists to provide a more versatile means of serializing entities. The entity serialization mechanism reads and writes XML documents with three high-level elements: `<Model/>`, `<View/>`, and `<Controller/>`. Each document includes one `<Model/>` element that describes the class and configuration of the model—minus its associated module configurations, which are represented by the separate `<View/>` and `<Controller/>` elements. Unlike a module configuration, which represents the configuration of its source model, an entity configuration represents the source entity itself. This means that one may create an entity and save its configuration as a prototype suitable for inclusion in other environments.

8. CONCLUSION

Techniques such as the definition of a consistent entity model and the use of well-known design patterns have helped the authors of the NPSNET-V component framework satisfy many of their stated requirements. The framework provides a simple and versatile platform for dynamically extensible networked virtual environments, suitable both for education and for advanced research. It makes use of existing standards such as HTTP and XML in order to maximize its interoperability with other systems, features a configuration and serialization mechanism able to support persistent virtual worlds, encourages modular application development, and provides a version-safe means of dynamically loading and upgrading components. Its efficiency and scalability, however, remain untested, as does its ability to facilitate interoperation between components and component configurations developed by different organizations.

Enabling that ability will be one of the primary goals of future NPSNET-V research. The NPSNET-V component framework is simply a foundation upon which to build; aside from its entity model, it provides few of the enormous number of conventions and interfaces that will be necessary to support the creation of massive shared virtual worlds from heterogeneous collections of components. There must be standards for networking, for graphics, for physical modeling; there must be ways of specifying user interfaces, of sharing resources, of integrating agent-based systems. In designing component-based architectures, one must often put as much effort into defining the interfaces that connect components as into the implementation of the components themselves. When correctly designed, however, component-based architectures justify their cost by providing the means to implement more flexible, more versatile, and more extensible applications.

The greatest benefits of a component-based architecture emerge when one combines it with an open source development model. By freely publishing the interfaces through which components interact, by providing component source code to learn from and build upon, and by supplying full documentation for each interface and each component, the designers of component-based architectures may draw upon the resources of the open-source community to create large and diverse libraries of interoperable components. To this end, the authors of the NPSNET-V component framework have established a presence at the popular SourceForge™ open source development web site. The NPSNET-V component framework, along with all source code written in support of NPSNET-V, may be downloaded from `http://sourceforge.net/projects/npsnetv`. It is the authors' hope that by encouraging the participation of the open source community in designing and developing NPSNET-V, they may allow it to grow from a simple research environment and educational toolkit into a platform capable of supporting the next generation of networked virtual worlds.

9. REFERENCES

[1] Berners-Lee, T.; Fielding, R.; Frystyk, H. *RFC 1945: Hypertext Transfer Protocol – HTTP/1.0.* Network Working Group, May 1996.

[2] Capps, M.; McGregor, D.; Brutzman, D.; Zyda, M. *NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments*. IEEE Computer Graphics and Applications, September/October 2000.

[3] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] Hubbold, R.; Cook, J.; Keates, M.; Gibson, S.; Howard, T.; Murta, A.; West, A. *GNU/Maverik: A Micro-Kernel for Large-Scale Virtual Environments.* Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 1999.

[5] Oliveira, M.; Crowcroft, J.; Slater, M. *Component Framework Infrastructure for Virtual Environments.* Proceedings of the ACM Collaborative Virtual Environments Conference, 2000.

[6] Pettifer, S.; Cook, J.; Marsh, J.; West, A. *DEVA3: Architecture for a Large-Scale Distributed Virtual Reality System.* Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 2000.

[7] Sun Microsystems, Inc. *Extensible Runtime Containment and Services Protocol for JavaBeans™.* L. Cable (Ed.),

1998. Available at http://java.sun.com/beans/glasgow/beancontext.pdf.

[8] Sun Microsystems, Inc. *JavaBeans™.* http://java.sun.com/products/javabeans.

[9] Watsen, K.; Zyda, M. *Bamboo – A Portable System for Dynamically Extensible, Real-Time, Networked Virtual Environments*. Proceedings of the IEEE Virtual Reality Annual International Symposium, 1998.

[10] Wilson, S.; Sayers, H.; McNeill, M.D.J. *Using CORBA Middleware to Support the Development of Distributed Virtual Environment Applications.* Proceedings of the WSCG International Conference in Central Europe on Computer Graphics, Visualization, and Computer Vision, 2001.