Data Coupling and Downcasting in Discrete Event Simulation Software*

James Nutaro, Richard Ward, Glenn Allgood
Oak Ridge National Laboratory
{nutarojj,wardrc1,allgoodgo}@ornl.gov

Alexander Parfenov, Jason Holmstedt
Physical Optics Corporation
aparfenov@poc.com , jtholmstedt@hotmail.com

Keywords: DEVS, discrete event simulation, object oriented simulation

Abstract

Discrete Event System Specification (DEVS) simulation libraries commonly make use of indirection and, essentially, typeless events as part of their interface specification. This forces library users to employ downcasting and/or strong data coupling in the design of their simulation applications. These techniques are anathema to good object oriented design principles, but seem to be inescapable when using pre-built DEVS simulation libraries. This paper describes how downcasting and data coupling emerge in the design of a computer architecture model. It is hoped that, by exposing the problem and its underlying causes, future research can be directed at improving software engineering techniques for DEVS simulation software.

1 INTRODUCTION

In object oriented software software systems, class hierarchies are constructed to represent variations on a common theme. When class hierarchies are designed well, it is possible to construct extensible software systems that rely only on information about base types (i.e., classes near the top of the hierarchy). This makes the software robust in the face of changes to specialized classes, resulting in code that is easier to maintain.

The animal farm is a classic example of object oriented programming. The farm has different types of animals, and every animal makes its own noise. Given the collection of animals on a farm, the objective is to print the noise that every animal makes. Suppose the farm has just a *Dog* and a *Cat*. Fig. 1 shows a C++ program that solves this problem by using downcasting.

This downcasting based solution is deficient, from an object oriented perspective, in two ways. First, adding new animals requires new **dynamic_cast** and **if** statements in the main for loop. Second, if the code encounters an unknown animal type, then it will fail to print the animal sound. These short comings are manifested in the need to downcast the *Animal* objects to specific *Dog* and *Cat* objects before the *makeSound* method can be called.

The need for downcasting can be overcome by using a virtual *makeSound* method in the *Animal* base class. With this new implementation, the two difficulties described above disappear; the need for downcasting is alleviated by the use of polymorphism. This solution does not require information specific to the derived types (i.e., *Dog* and *Cat*), and so the need for downcasting is eliminated. The revised code listing is shown in Fig. 2.

Discrete Event System Specification (DEVS) simulation APIs commonly include a base class from which input and output event classes are derived (see, e.g, [2], [7], [1], and [3]). While models operate on the derived event classes, the simulation engine is aware only of the base class. Consequently, type information is lost when an event passes from a model to the simulation engine and back.

The use of indirection and, essentially, typeless events

```
class Animal {
    public:
        virtual ~Animal(){}
};
class Dog: public Animal {
    public:
        void makeSound() {
            cout << "WOOF!" << endl;
    }
};
class Cat: public Animal {
    public:
        void makeSound() {
            cout << "MEOW!" << endl;
        }
};

int main() {
    Animal* Farm[2];
    Farm[0] = new Dog; Farm[1] = new Cat;
    for (int i = 0; i < 2; i++) {
        Dog* dog =
            dynamic_cast<Dog*>(Farm[i]);
        Cat* cat =
            dynamic_cast<Cat*>(Farm[i]);
        if (dog != NULL)
            dog->makeSound();
        if (cat != NULL)
            cat->makeSound();
    }
    return 0;
}
```

Figure 1. Solution with downcasting: an Animal farm without polymorphism.

in the core simulation engine leads to one of two design choices for applications. The first is to use a single super-event for model input and output. This creates strong data coupling in the application. The second is to use run-time type identification and downcasting to properly interpret input events.

The super-event solution introduces a single class that encompasses all possible event types. This solves the unknown type problem by permitting only one type. While input events are downcast before being processed, the downcasting is always safe because there is only one possible target type.

The second solution relies extensively on downcasting and run-time type identification to process input events.

```
class Animal {
    public:
        virtual void makeSound() = 0;
        virtual ~Animal(){}
};
class Dog: public Animal {
    public:
        void makeSound() {
            cout << "WOOF!" << endl;
        }
};
class Cat: public Animal {
    public:
        void makeSound() {
            cout << "MEOW!" << endl;
        }
};

int main() {
    Animal* Farm[2];
    Farm[0] = new Dog; Farm[1] = new Cat;
    for (int i = 0; i < 2; i++)
        Farm[i]->makeSound();
    return 0;
}
```

Figure 2. Solution without downcasting: an Animal farm with polymorphism.

While this avoids the worst aspects of data coupling, it introduces an implicit dependence between communicating components. In this case, the event generating component must produce an output type that is compatible with the expected input type of the receiving component. Violations of this implicit agreement can cause the simulation software to fail. Moreover, incompatibility problems only appear at run-time, and so the overall robustness of the software is necessarily diminished.

A general principle of object oriented software is that the need for downcasting indicates a design flaw. However, it appears that current approaches to object oriented, DEVS-based simulation do not, or possibly can not, effectively use polymorphism in their event processing routines. By using a super-event to avoid downcasting, the designer trades downcasting for data coupling. Neither solution is attractive, but no alternatives seem to be available.

This paper documents the use of data coupling and downcasting in an object oriented computer architecture model. The model was developed for the Physical Optics

Corporation to support performance related design studies. It is hoped that, by exposing the source of the design problem and its underlying causes, future research can be directed at improving software engineering techniques for DEVS simulation software.

2 THE MODEL

The model is used to assess the performance of a proposed computer architecture for specific supercomputing applications at Oak Ridge National Laboratory. A performance assessment is done by executing an abstract model of the application on the simulated computer. The computer model captures throughput characteristics of the component hardware, first in/first out queuing of requests by device drivers, and it can execute programs described as a sequence of send and receive operations, disk access operations, and CPU bound calculations.

The computer described by the model has four compute nodes that are linked by a high speed optical network. The model is decomposed into three layers. The top layer is the complete four node system. At the middle layer are the distinct nodes and the optical network. The network model is an abstract representation of the Physical Optics Corporation optical interconnect. The network model captures salient performance characteristics, which include the anticipated network latency and throughput characteristics. This mid-level view of the model is shown in Fig. 3.

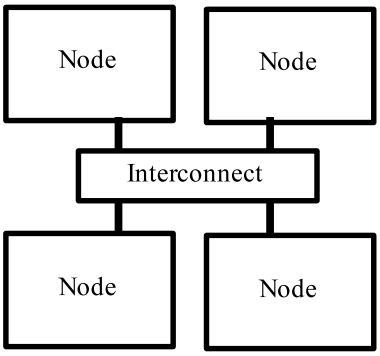

Figure 3. Node and interconnect view of the model.

A node model is decomposed into its component parts, as shown in Fig. 4. The PCI-X bus, HyperTransport tunnel, and optical network adapter model describe specific devices and their driver software. These models capture data transport latency as a function of data size and device throughput. In addition to this basic function, the PCI-X bus and HyperTransport tunnel have data routing capabilities. The PCI-X bus can deliver data to targeted PCI devices. The HyperTransport tunnel can forward data to the PCI-X bus or communicate directly with the SCSI disk controller.

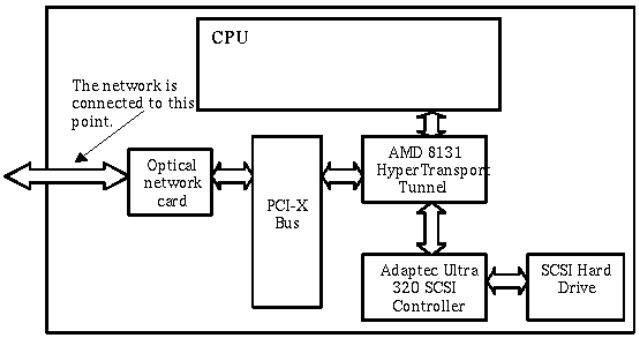

Figure 4. Block diagram of the node model.

The disk drive, SCSI disk controller, and CPU model execute requests. In the case of the CPU model, these requests are a sequence of program instructions (i.e., network send, network receive, disk read, disk write, and compute). An example of a program that can be executed by the CPU model is shown in Fig. 5. The network instructions require communication with the optical network adapter via the PCI-X bus and HyperTransport tunnel. The disk read and write instructions are serviced by communicating with the SCSI disk controller via the HyperTransport tunnel. The SCSI controller, in turn, executes requests to access the disk. A disk access request begins a series of interactions between the SCSI controller and the disk drive that model the disk access activity.

```
### Program for Node 1 ###
### Repeat for some number of time steps ###
repeat 4000
        ### Exchange edge cells with adjacent nodes ###
        send 1000 2
        send 1000 3
        recv 2
        recv 3
        ### Compute local grid point values ###
        compute 100.0
end
### Accumulate results ###
recv 2
recv 3
recv 4
### Do post processing on results ###
compute 100.0
### Dump the results to a file ###
disk_write 1000
```

Figure 5. Program listing that can be executed by the CPU model.

The model output shows the cumulative times that each CPU model spent performing its four major activ-

ities. These activities include time spent waiting to receive data from the network and disk, time spent writing to the disk, and the amount of CPU bound computation. Figure 6 shows the output data for a CPU. The model produces four such output files, one per CPU.

```
recv_wait_time 2956.04
disk_read_time 0.00778285
disk_write_time 0.00759405
cpu_time 6.31976e+06
```

Figure 6. Model output file for one CPU.

3 EVENT OBJECTS IN ADEVS

The model is implemented in C++ using the *Adevs* simulation engine (see [3]). *Adevs* uses a class called *object* as the base class for all events. Events are routed through a multicomponent model using *PortValue* objects. Each *PortValue* object has two fields; a port identifier and a pointer to an object of type *object*. The simulation engine and models exchange events using dynamic arrays, implemented by the *adevs_bag<PortValue>* class, of *PortValue* objects.

Input events are delivered to a model via two virtual methods that correspond to the external and confluent transition functions of an atomic DEVS model. These methods are implemented by every model, and they are called by the simulation engine whenever input events are available for the model to process. Fig. 8 shows an abbreviated listing of the external transition function for the HyperTransport model. The *NodeEvent* class that appears in the code listing is derived from the *object* class.

When a model changes state autonomously, the simulation engine calls a virtual method that corresponds to the atomic model output function. This method accepts an empty bag, and fills it with *PortValue* objects that represent the model output events. The output function for the HyperTransport tunnel model is shown in Fig. 7.

4 DATA COUPLING

Data coupling (sometimes called stamp coupling) occurs when software components share a complex data type. Data coupling can appear when an excessive number of

```
void AMD_8131_HyperTransport_PCI_X_Tunnel::
output_func(adevs_bag<PortValue>& y) {
    // Send the first request in the queue
    //to the appropriate output port.
    output(req_q.front().dst,
        req_q.front().e.clone(),y);
}
```

Figure 7. The HyperTransport tunnel output function.

arguments, typically in the form of a large data structure, are accepted as a method (or function) argument. When some of these arguments are unnecessary, the interface becomes difficult to use and understand (see, e.g., [4]).

To see how data coupling occurs in the computer architecture model, consider two possible interactions. The first is the creation of a disk read request by the CPU. The information needed by the SCSI controller to process a read request are the request type and the number of bytes to read. This information appears in the program instruction that is executed by the CPU, and so it can be easily provided by the CPU model.

As the request moves from the CPU to the SCSI controller, it passes through the HyperTransport tunnel. The HyperTransport tunnel needs to know the number of bytes being moved and the location of the target device (i.e., on the PCI-X bus or the SCSI controller). This information is also provided by the CPU.

The data flow for processing a disk read request is summarized in Fig. 9. The CPU provides the HyperTransport indicator, data size, and request type. This information passes first through the HyperTransport tunnel model, which requires only the HyperTransport indicator and data size. It then passes from the HyperTransport tunnel to the SCSI controller, which requires only the request type and data size.

A simple implementation of this data flow, and the one used in this model, is to create a super-event that contains all of the information needed to route the request through the intermediate hardware and process it on arrival at the target device. A *NodeEvent* class is created to hold all of the input and output data required by the models. In this particular case, the *NodeEvent* contains the request type, HyperTransport indicator, and data size.

This creates the first instance of data coupling. The HyperTransport tunnel receives unnecessary data in the form of the request type. The SCSI controller receives the HyperTransport indicator, which is not useful to it in

```
void AMD_8131_HyperTransport_PCI_X_Tunnel::delta_ext(double e, const adevs_bag<PortValue>& x)
{
    // Add incoming requests to the request queue.
    for (int i = 0; i < x.getSize(); i++) {
        ht_msg_t msg;
        msg.src = x.get(i).port;
        const NodeEvent* e = dynamic_cast<const NodeEvent*>(x.get(i).value);
        msg.e = *e;
        // Route the message to the appropriate output port.
        if (msg.src == side_A_in && e->protocol == HYPER_TRANSPORT_PROTOCOL) {
            msg.dst = side_B_out;
            req_q.push_back(msg);
        }
        else if (msg.src == side_B_in && e->protocol == HYPER_TRANSPORT_PROTOCOL) {
            msg.dst = side_A_out;
            req_q.push_back(msg);
        }
        // .....
        else if (msg.src == pci_bus_B_in && e->protocol == PCI_X_PROTOCOL &&
        msg.e.target_device_addr == pci_addr_b) {
            msg.dst = side_B_out;
            req_q.push_back(msg);
        }
        // If this is the first request, then process it
        if (req_q.size() == 1 && timeNext() == ADEVS_INFINITY) {
            process_next_req();
        }
    }
}
```

Figure 8. The HyperTransport tunnel external transition functions.

any way. Access to these unnecessary data items introduces an opportunity for errors. For instance, if the HyperTransport tunnel does not preserve the request type, then the disk read request will fail.

The problem grows as new features are added to the model and, consequently, more data is added to the super-event. Fig. 10 shows the *NodeEvent* class that is used in the complete model. Although each data field is required for some transaction within the model, only a handful (two or three fields) are required by any particular transaction. Moreover, as the number of intermediate components that participate in a transaction grows, the potential for mishandling data grows as well. Possible mishandling includes inconsistent use of a data item or the accidental introduction of erroneous values when events are copied or modified.

5 DOWNCASTING

Suppose a component is given an object of class A, but it needs an object of class B. If the class B is derived from class A, it is possible to downcast the object class from A to B. In practice, downcasting takes the form of a query and a cast. Is this object of class B derived from class A? If it is, then cast the object as one of class B.

Because the simulation engine is aware only of the base class, it is necessary for models to downcast input events before they are processed. Downcasting can be done blindly when a single event class is allowed. In designs that allow for multiple event classes, the appropriate cast must be determined using the port identifier or the language run-time type identification system. In the first case, failures due to downcasting are avoided at the expense of introducing data coupling. In instances where multiple event classes are permitted, unexpected

```
class NodeEvent: public object {
    public:
        NodeEvent(object* user_data = NULL);
        NodeEvent(const NodeEvent& src, object* user_data);
        NodeEvent(const NodeEvent& src);
        const NodeEvent& operator=(const NodeEvent& src);
        ~NodeEvent();
        // Create a copy of this object.
        object* clone() const;
        // Communication protocol and device used to transport this event.
        node_protocol_t protocol;
        // Target address for a message. Addresses are protocol dependent.
        int target_device_addr;
        // Originating address for a message. Addresses are protocol dependent.
        int orig_device_addr;
        // Device specific request type.
        device_request_t req_type;
        // Number of bits comprising the request.
        int num_bits;
        // Get any user data that was assigned to this event.
        const object* get_data() const;
};
```

Figure 10. The *NodeEvent* class definition.

failures can occur when the class of an input event is unknown or unexpected.

In the computer architecture model, problematic downcasting is avoided by using a single super-event class (i.e., the *NodeEvent* class). However, to illustrate a downcasting based solution, suppose that each component model defines its own event classes, and these classes carry only the information that is needed by the component. For the interaction described previously, there are three distinct event types. A *DiskRequestEvent* is defined for the SCSI controller, and its attributes are the request type and data size. A *HyperTransportEvent* is defined for the HyperTransport tunnel, and its attributes are the destination of the data transmission, a reference to the data being transported, and the data size.

To initiate a disk read request, the CPU creates a *HyperTransportEvent* whose transported data reference attribute points to a *DiskRequestEvent*. This is received by the HyperTransport tunnel as an *object* that must be downcast to a *HyperTransportEvent*. The HyperTransport tunnel then passes the *DiskRequestEvent* to the SCSI controller, where it is received as an *object* that must be downcast to a *DiskRequestEvent* before being processed.

This solution could fail for several reasons. In one scenario, the event source has misunderstood the input requirements of the event destination. This could occur, for instance, because of an error in the component documentation. In another scenario, the event source uses an obsolete interface definition (e.g., an obsolete header file in conjunction with an up to date version of the object code). In a third scenario, the coupling constraints are misunderstood by the component integrator. For example, connecting the CPU directly to the SCSI controller will cause a run-time failure.

The origin of these failures are difficult to find. They occur at run-time because the compiler is unable to perform type checking on model input and output events. Moreover, an event moves by an indirect path from its source (i.e., the CPU) to its destination (i.e., the SCSI controller).

6 CONCLUSIONS

The widespread use of a base class for event representation in DEVS simulation engines suggests that data coupling and downcasting are common features of DEVS based simulation applications. The computer architec-

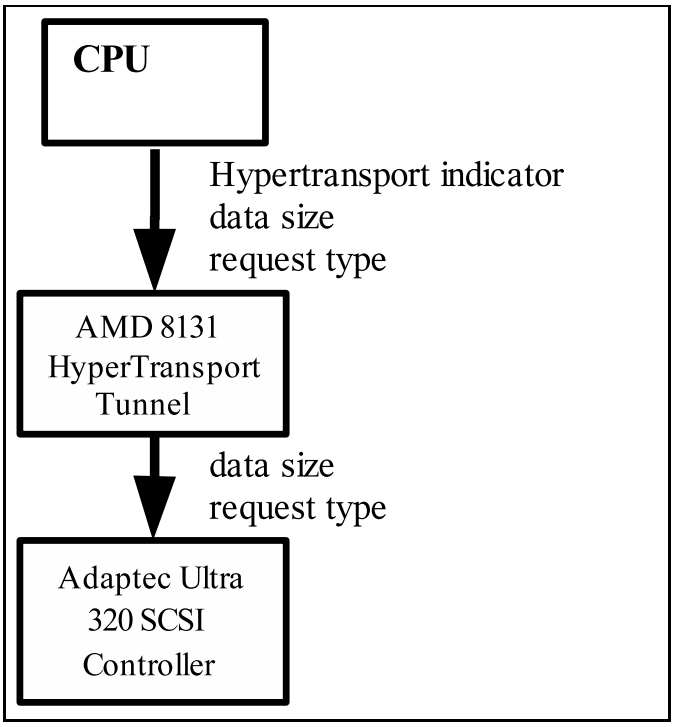

Figure 9. Data flow for a disk read request.

ture model described in this paper is a stereotypical example of a discrete event model, and so the design issues that emerge in this application are likely to appear in many DEVS modeling projects.

It is difficult to avoid data coupling and extensive use of downcasting given a simulation engine that uses indirection and a base class for event representation. A solution to this problem might require new design patterns for DEVS simulators, but promising candidates have yet to be identified. The use of coupling compilers (see, e.g., [2] and [5]) and application specific languages (see, e.g., [6]) might suggest another solution. Special languages for describing model coupling could, for instance, hide an underlying super-event solution by automatically creating appropriate class aliases (e.g., C typedefs) and enforcing access constraints at compile time.

The problems of downcasting and data coupling can be avoided in simulation software based on the event scheduling paradigm. Within this paradigm, a context is included with every event handler. The context object is used in the event handling routine to implement changes in its own state. Figure 11 sketches an event scheduling solution to the disk request example described above. A similar approach, adapted to the interface requirements of DEVS based simulation libraries, might also provide a solution to the software engineering problems described in this paper.

References

[1] Jean-Baptiste Filippi and Paul Bisgambiglia. JDEVS: an implementation of a DEVS based formal framework for environmental modelling. *Environmental Modelling & Software*, 19(3):261–274, March 2004.

[2] E. Kofman, M. Lapadula, and E. Pagliero. PowerDEVS: A DEVS-Based Environment for Hybrid System Modeling and Simulation. Technical Report LSD0306, School of Electronic Engineering, Universidad Nacional de Rosario, Rosario, Argentina, 2003.

[3] A. Muzy and J. Nutaro. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling & Simulation*, pages 401–407, ISIMA / Blaise Pascal University, France, June 2005.

[4] Roger S. Pressman. *Software Engineering: A Practitioner's Approach, Sixth Edition.* McGraw-Hill, New York, New York, 2005.

[5] András Varga. OMNeT++ user manual, OMNeT++ version 3.2. Unpublished manuscript, 2005.

[6] Gabriel Wainer. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

[7] Bernard P. Zeigler and Hessam S. Sarjoughian. Introduction to DEVS modeling and simulation with Java: Developing component-based simulation models. Unpublished manuscript, 2005.

```
class NodeEvent;

class AMD_8131_HyperTransport_PCI_X_Tunnel {
    public: // Event processing methods
                void handleNodeEvent(NodeEvent* e);
                // ...
    private: // State variables
                deque<ht_msg_t> req_q;
                // ...
                void process_next_req();
};
class NodeEvent: public SimulationEvent {
    public: // Event specific data is the same as before
                int src;
                node_protocol_t protocol;
                // ...
                // Event context
                AMD_8131_HyperTransport_PCI_X_Tunnel* context;
                // Event handler
                void handler() { context->handleNodeEvent(this); }
};
void AMD_8131_HyperTransport_PCI_X_Tunnel::
handleNodeEvent(NodeEvent* e) {
    if (e->protocol == HYPER_TRANSPORT_PROTOCOL && e->src == side_A_in) {
        ht_msg_t msg;
        // Create message from NodeEvent e
        req_q.push_back(msg);
    }
    // ...
    process_next_req(); // Schedule another NodeEvent
}
```

Figure 11. Sketch of an event scheduling implementation for the HyperTransport tunnel.