

# Colorado Technical University



## Technical Report Computer Science

### **A simulation framework with direct support for associations**

Charles A. Suscheck  
TurboPower Software  
Colorado Springs  
charless@turbopower.com

Bo I. Sandén  
Professor of Computer Science  
Colorado Technical University  
bsanden@coloradotech.edu

**Technical Report Number  
CTU-CS-2001-002**

# A simulation framework with direct support for associations

Charles A. Suscheck  
TurboPower Software  
Colorado Springs  
charless@turbopower.com

Bo I. Sandén  
Colorado Technical University  
Colorado Springs  
bsanden@acm.org

## ABSTRACT

*Many existing frameworks for discrete event simulation impose a structure on the implementation that distorts traceability between it and the conceptual domain model. In particular, events and constructs must be added to support associations. Typically, thousands of associations form and dissolve during a simulation run, so their manipulation is critical. This paper presents a simulation package called EASY that specifically supports association manipulation with built-in mechanisms that reduce the need for implementation constructs. EASY promotes loose coupling between domain objects and improves the traceability between conceptual models and their implementations. The design can be incorporated into other simulation frameworks to simplify the implementation and allow it to reflect rather than distort the domain model.*

## 1. INTRODUCTION

Expectations for accurate computer simulations are greater today than at any other time. In many areas, such as storm tracking, military engagement, traffic flow, genetic engineering, chemical engineering, and space flight, simulations are mission or safety critical. Increasing the accuracy of a simulation generally means adding details to the domain model. As a result, the complexity of the domain becomes difficult to manage and the performance of the system suffers.

Techniques exist that greatly enhance the performance of a simulation. One technique that takes advantage of high performance processors is optimistic processing, where events are processed faster than wall clock time and then made available to the simulation. The idea is to process as many events as possible before they exceed their allocated wall clock time so that events that do exceed their allocated time will not bog down the simulation [13]. A problem is that an optimistically processed event may be negated by later happenings. Processing must then be rolled back and a new sequence of events executed. This occurs for example in human-in-the-loop simulations, where a human participant may insert an event.

Parallelism is a further refinement of optimistic processing. Events are distributed among processing nodes either on a network or within a multiple-CPU machine. Optimism combined with parallelism complicates roll back. Nonetheless, the benefit of using a network of processors is that more processing time is available.

Optimism and parallelism increase the performance at the cost of added complexity. Simulation frameworks are a way to reduce this complexity. They hide the simulation operation from the domain-model implementation and interact with the domain by application programmer interfaces, inheritance, events, composition, or other such constructs. Another way to reduce complexity is to use object orientation. OO techniques were specifically developed to reduce domain complexity. They model the user's perspective of the system in a semantically meaningful manner, which, most significantly, follows human conceptualization. "Object-oriented systems allow the real world to be represented more directly than do conventional ones" [5]. A number of object-oriented parallel discrete event simulation (OO-OPDES) frameworks exist. Such a framework can hide the complexities of performance enhancing techniques while allowing modelers to capture the domain in terms of objects and classes with methods representing the events that involve each class.

Existing OO-OPDES frameworks have an unfortunate effect on the conceptual domain during implementation. They generally impose a foreign structure on the implementation by adding constructs that do not necessarily reflect the conceptual domain model. The main problem is that events become first class objects and distort the operation of the implementation to the point where it no longer reflects the conceptual model. Events rather than domain objects become the preeminent design consideration.

The PDES community appears to play down the role of the domain model [10, 16]. This puts simulation frameworks at odds with proponents of object-oriented techniques, who stress a semantically meaningful model of the system as one of the key points in communicating the design between users and developers. Less impact by the simulation package leads to a better mapping between the conceptual domain model and the implementation model. Better mapping has distinct advantages. There is less opportunity for misunderstanding between the system developer and the domain expert. Changes in the conceptual model are easier to implement. Fewer mechanistic implementation classes reduce the chance that errors will be introduced in such classes.

The distortion of the conceptual model becomes particularly apparent where associations between domain objects are involved. Dynamically formed and dissolved associations are an important part of the domain model in many simulations. For example, the phenomenon "radar detects missile" creates an association between a radar and a missile object. Current OO-OPDES frameworks leave the manipulation of associations to the modeler, who must add implementation classes and events to the simulation model in order to support them.

This paper proposes an OO-OPDES package called *Event Association Simulation Package (EASY)* that focuses on associations [15]. It supports domain model implementations that reflect rather than distort the conceptual model. EASY hides the implementation classes and events that must be included to manipulate associations. Through junction objects, container

classes and support mechanisms specifically designed to manipulate associations, it separates the dynamic association-based interactions of the domain objects from their context-independent behavior. This separation lets the domain modeler concentrate on the behavior of the domain and leads to a cleaner, more intuitive, and more reusable implementation. EASY makes the following contributions:

1. The simulation framework supports the creation and manipulation of associations by means of classes that the simulation implementer extends.
2. Design patterns are used, and techniques to foster better OO design are incorporated into the way in which events are handled.

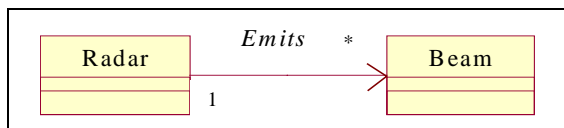
EASY introduces less complexity into a domain model than representative simulation packages because it supports real world constructs in a straightforward fashion. Mechanisms such as the event propagation techniques and cascading events are key components. These mechanisms can also be incorporated in other OO-PDES frameworks.

## 2. RELATED WORK

### 2.1 Associations

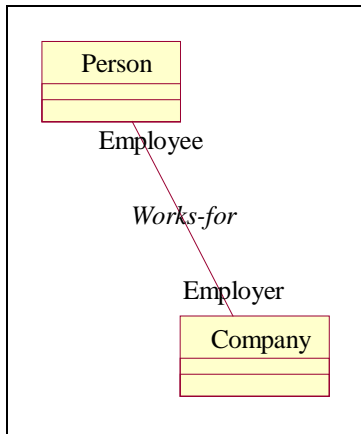
An association captures a semantic relationship among classes in the domain. Associations are supported in nearly all object-oriented notations. A *binary* association between classes B and C represents a set of links between B instances and C instances. Each link is defined by the two instances so an association cannot include more than one link between a given B instance and a given C instance. In a *self-association*, classes B and C are identical. A *ternary* association defines links between instances of three or more classes.

An association is by default bidirectional meaning that it can be read from either end with significance. In Fig. 1, a radar emits a beam, and the beam is emitted by a radar. Adding an arrowhead at one end specifies that the association is only *navigable* in one direction. Given a Radar object, the associated Beam objects can be identified, but a Beam object has no reference to the Radar object emitting it. Fig. 1 also shows *multiplicity*, which constrains the number of related objects: A radar emits zero or more beams and each beam comes from exactly one radar.



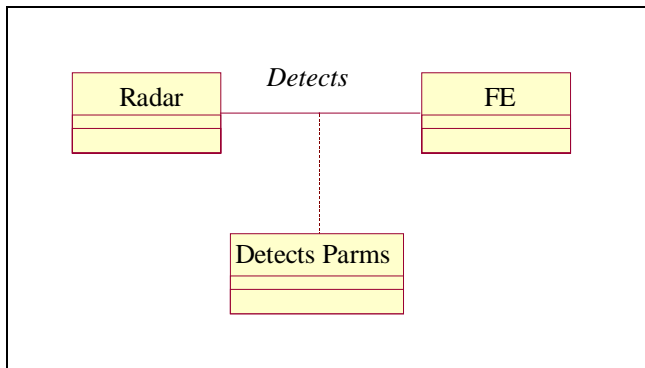
**Figure 1** An association with direction

A *role* is the function, behavior, or assigned characterization that an object has in an association [1, 16]. In Fig. 2, a person plays the role of employee while the company plays the role of employer in the works-for association. Role names are sometimes used instead of association names.



**Figure 2 Roles in an association**

Properties can be attached to an association by means of an *association class* [1, 11]. An association class has exactly one instance for each set of objects linked through the association. If a link is dissolved, the association class instance is destroyed. In Fig. 3, Radar detects a flying entity. The DetectsParms association class can contain such information as time of detection, radar cross section, and signal strength, which are only relevant to a particular radar and flying entity pair.



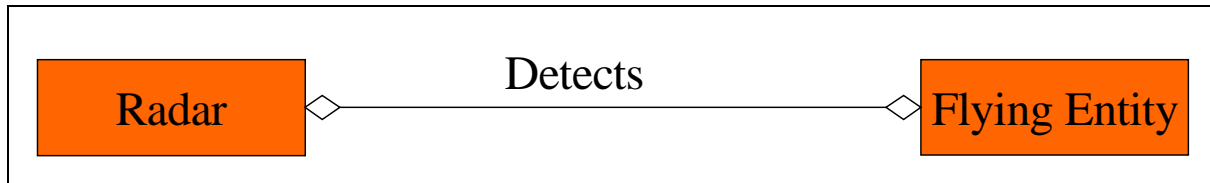
**Figure 3 An association class, DetectsParms**

### 2.1.1 Advanced association concepts

Researchers have introduced association concepts that are more elaborate than those discussed here. *Role based behavior* [16], means that the behavior of an object can change depending on the role it plays. When an association is formed between two instances, the behavior of the associated instances is altered in some way. A real world example is a person who becomes a parent. The person has a parental association with a young person (a child) and the behavior of the person is changed due to this association. Another example is a radar that has detected a flying entity. The radar now enters a tracking phase where it moves to follow the detected entity.

### 2.1.2 Implementation of associations

Associations can be implemented by means of one way or two-way pointers (direct containment) or by additional object constructs [8]. Fig 4 shows containment with two-way pointers. A diamond is used to indicate the root of a pointer. If a radar detects multiple flying entities, the Radar object will contain an array of pointers to flying entities, and each flying entity will contain a reciprocal pointer to the radar.



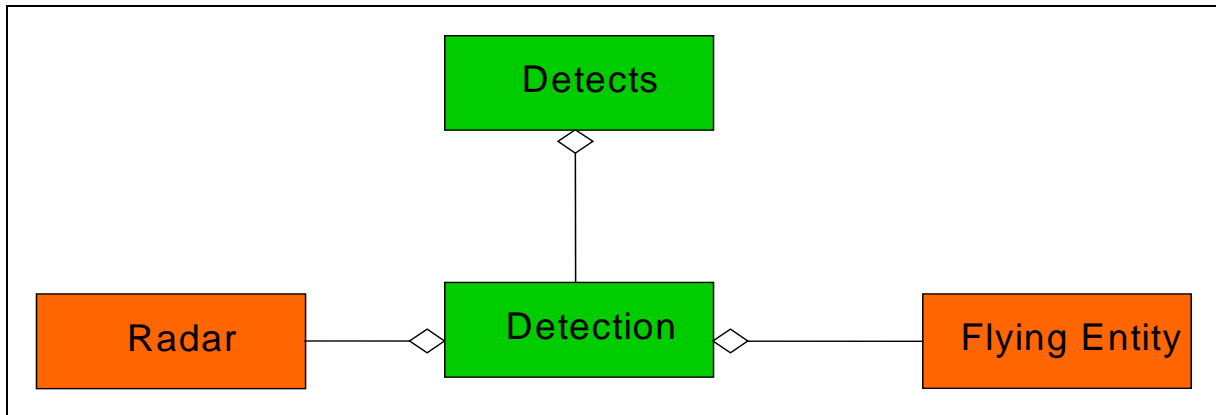
**Figure 4 Using bi-directional pointers to implement associations**

Although easy to implement, direct containment has problems with extensibility. Adding a new association to a class means adding a new array of pointers. If an association is no longer viable, an empty pointer array is still contained within the class. Another problem is the potential for referential integrity rifts. In Fig. 4, if the radar is deleted, the pointer in the Flying Entity must be updated to no longer reference it. Finally, association classes are difficult to represent using direct containment. Either a new pointer to the association class instance must be created, or its properties must be captured within the participating classes, violating the domain semantics.

### 2.1.3 Using constructs to implement associations

As pointed out in [16], associations should be treated as first-class entities. They should not be buried inside objects since they are not subordinate to a particular object but depend upon two or more classes. “Some information inherently transcends a single class, and the failure to treat associations on an equal footing with classes can lead to programs containing hidden assumptions and dependencies.” [11]. With an explicit object representing each link between instances, the modeler can peer into the operation of the simulation and answer questions that allow validation and verification of the model as well as analyze the results of the simulation.

Some of the difficulties with bi-directional pointers can be alleviated if associations are implemented using *junction classes* and *container classes* [3]. Each instance of a junction class has a one directional pointer to each object linked by the association. A container object represents a set of junction objects. In Fig. 5, Detection is a junction class and Detects is a container class. A Detection object has pointers to a Radar instance and a Flying Entity instance. A Detects object has pointers to many Detection instances.



**Figure 5 Using constructs to implement an association**

Using constructs to implement associations has several advantages. Relationships can be added without changing the domain classes. “This advantage is absolutely critical for large-scale software reuse; otherwise objects need to change every time they are used in a different application” [16]. Furthermore, the container classes can be used for instance accounting, and designers can apply object-oriented techniques to the container classes themselves.

One or more association classes can easily be added to the association, and higher order associations can be modeled by adding more pointers to the junction object. Another advantage is that queries such as “what radars detect any flying entities at time T” can be easily answered. This is particularly important when users follow the progress of the simulation on a GUI. If the associations between the domain objects are buried within the domain objects, the queries must be methods of the domain classes, which pollutes the semantics. For example, a radar that detects a flying entity might not know its identity, so querying on a particular entity may be invalid from a domain perspective.

Containers are artificial implementation classes, but EASY compensates for this drawback by encapsulating the manipulation and structure of associations. Simply adding an association construct to the domain model will detract from the semantic traceability between the implementation and the domain model. EASY includes mechanisms specifically intended to hide these complexities, so associations can be added with little effect on the domain model.

## 2.2 Simulation packages

Simulation systems are traditionally based on one of the three world views: event scheduling, activity scanning, and process interaction [2]. These views have been described as follows:

“Event scheduling provides locality of time. Each event routine describes related actions that may all occur in a single instant. Activity scanning provides locality of state. Each activity routine describes all actions that must occur because a particular model state is reached. Process interaction provides locality of object. Each process routine describes the entire action sequence of a particular model object” [9].

Event scheduling is common and is the perspective of EASY. In an event-scheduling simulation, activity is precipitated by events, and event-scheduling simulation packages can be classified according to the effect of each event as follows:

In a state manipulation simulation package, events cause the state of the simulation to change. This perspective is very similar to the declarative model [2]. While other processing also occurs, the primary concern of the simulation modeler is to determine where state changes take place and schedule the transitions as events. Events are dispatched by the simulation engine and act upon the domain objects by modifying their state.

A process oriented simulation package focuses on the processing that takes place upon the occurrence of an event. This is similar to the functional model [2]. Events are dispatched by the simulation engine and act upon the domain objects by calling their methods.

An association oriented package [15] focuses on association manipulation. The domain objects release each event to the simulation engine, which forms an association with the appropriate instances under certain constraints. EASY is an association-oriented event-scheduling simulation framework. It has specific mechanisms to encapsulate the manipulation of associations.

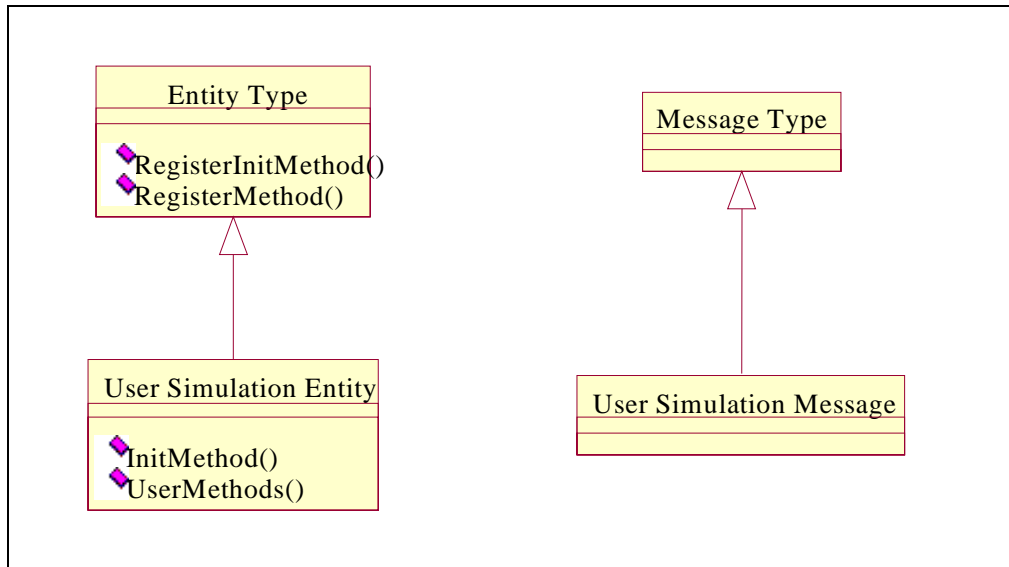
Many existing simulation packages were built for specific hardware or a particular class of simulations. COMPOSE [6], SPEEDES [12, 13, 14], IMPORT [16], and WARPED [7] are hardware independent, general-purpose packages targeted to object-oriented domain models. SPEEDES is a state manipulation package where state exchange between the event and the domain objects is the primary effect of an event. COMPOSE is a process oriented package. It links methods to messages representing events, causing a process to occur when the event fires. COMPOSE and SPEEDES are discussed in some detail in the following. The operations of WARPED and COMPOSE are very similar, but COMPOSE is more thoroughly documented. IMPORT was discarded from this discussion because it is in some places incomplete.

No simulation is strictly state, process, or association oriented. SPEEDES calls methods when an event occurs, and COMPOSE can be used in such a manner as to exchange state, as SPEEDES does automatically. EASY has properties of both SPEEDES and COMPOSE.

### **2.2.1 A process oriented simulation package: COMPOSE 1.0**

COMPOSE (Conservative, Optimistic and Mixed Parallel Object-oriented Simulation Environment) operates by scheduling events (implemented as message classes) that call methods on the domain objects [6].

COMPOSE provides the two main classes: *EntityType* and *MessageType* (Fig. 6). Domain objects must inherit from *EntityType*, which contains methods such as *RegisterInitMethod* and *RegisterMethod* that enable parallelism and distribution of the entities. User simulation messages must inherit from *MessageType*, which includes timestamp and source/destination information that is used by the COMPOSE engine.

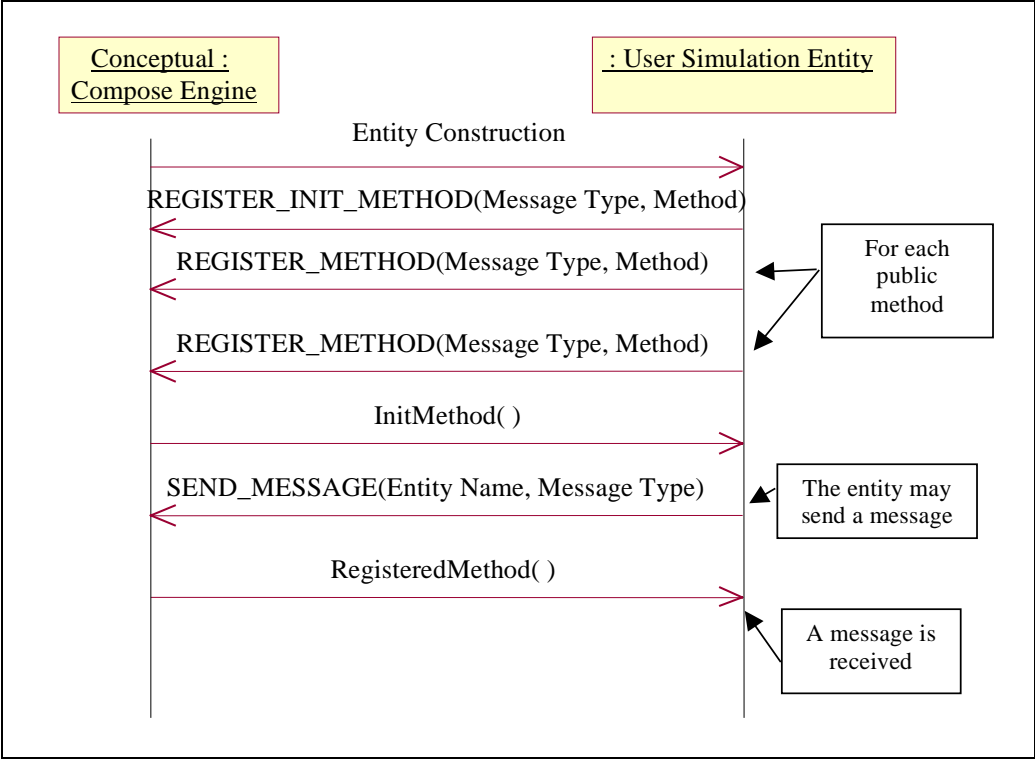


**Figure 6 Inheritance relationship between user entities and COMPOSE base classes**

When an event is due to happen, the COMPOSE engine dispatches the corresponding message to appropriate processing nodes. All methods that are registered for that message type, T, say, are then invoked. For example, any number of simulation entities can have methods registered for T, but each entity can have at most one method registered for T. Every public method of a simulation entity must be registered with exactly one corresponding message class. The message contains the parameters associated with the method calls.

Fig. 7 shows the typical operation sequence for a simulation entity. When a simulation object is constructed, it registers its initialization method with REGISTER\_INIT\_METHOD and then registers each public method for a message class with REGISTER\_METHOD. The initialization method is called immediately after the simulation entity has been constructed. During system initialization, the simulation entity may send a message to itself or other simulation entities.

When a message is dispatched by the COMPOSE engine, the registered methods for the message is called. The message contains the data used by the methods. Any time a method is invoked via a message, it is executed to completion without interruption.



**Figure 7 Example of a simulation entity sending messages**

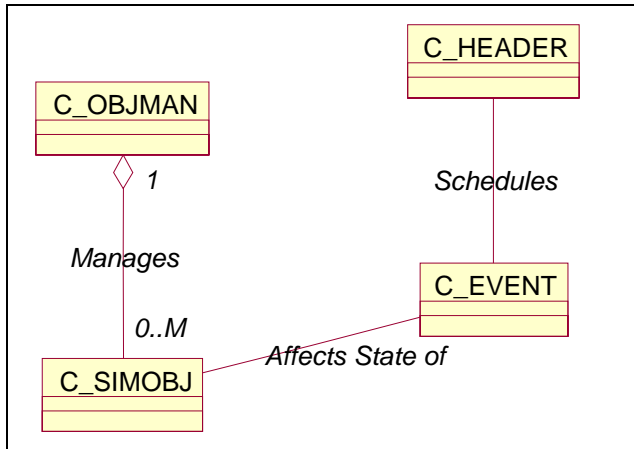
COMPOSE allows objects in the simulation to dynamically switch execution modes between optimistic and conservative algorithms, which is an unusual capability. The modes are defined per object. The simulation can execute with a mixture of optimistic and conservative algorithms controlled at a fine level of granularity.

**2.2.2 A state manipulation simulation package: SPEEDES 0.3**

The philosophy of SPEEDES 0.3 (Synchronous Parallel Environment for Emulation and Discrete Event Simulation [12]) is that simulation objects are data stores of state. SPEEDES is widely used for military simulations. It allows the simulation engineer to switch between a variety of conservative and optimistic event processing algorithms without changing the simulation model. Unlike COMPOSE, the granularity is at the system level and the algorithms must be selected at startup.

A state manipulation simulation package implements communication between objects by means of events, whose main purpose is to modify state. In SPEEDES, simulation objects have no behavior and are only acted upon by events. An event contains data representing a state change and can query the simulation objects and exchange data with them. The event's data is then pushed, field by field, into the domain object, and the data from the domain object is saved within the event. If a rollback occurs, the same exchange mechanism is used to restore both the domain object and the event object.

The domain modeler must determine the state of the domain object and when the state must take effect. The state is then captured in an event, which is scheduled by a message. A domain object inherits from one of four base classes: the simulation object base class C\_SIMOBJ, the object manager base class C\_OBJMAN, the message base class C\_HEADER, and the event object base class C\_EVENT (Fig. 8).



**Figure 8 SPEEDES class relationships**

An object manager (subclass of C\_OBJMAN) contains and manages multiple simulation objects. Object managers assign simulation objects to the different processing nodes and gather end-of-simulation statistics. Each different subclass of C\_SIMOBJ requires an object manager instance. SPEEDES calls a method for each object manager to schedule the initial events for the simulation objects it contains. Object managers are simulation objects, so events can act upon them. Such events can affect all the domain objects contained in the object manager.

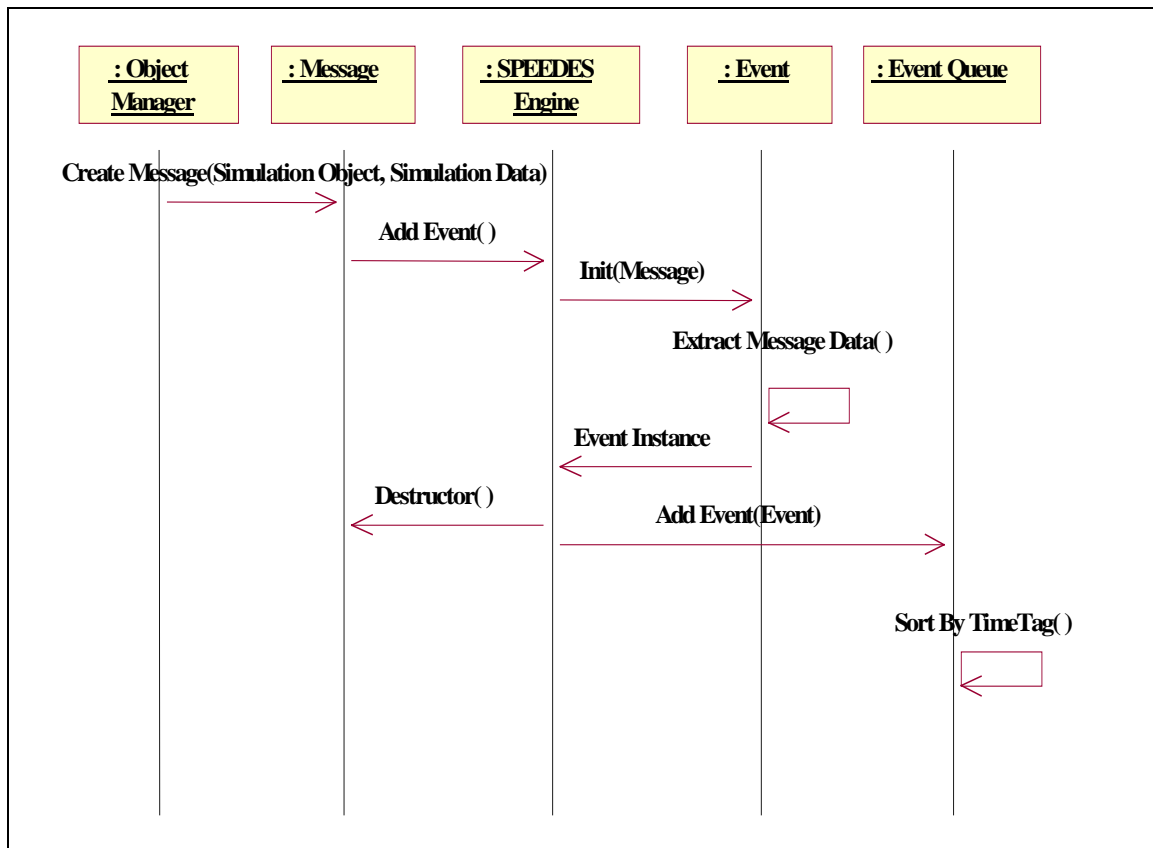
Simulation objects (subclasses of C\_SIMOBJ) are the domain objects. Data within simulation objects is modified by the EXCHANGE method defined in the C\_EVENT class. Event objects do the scheduling and state modifications so that the simulation objects are not coupled to the simulation.

A message (subclass of C\_HEADER) is used to create events. It contains a timestamp, data to be exchanged with the simulation object, and a reference to the simulation object. Messages are stored in the simulation event queue. When the simulation time advances to the point that the message is valid, an event is created and the message is destroyed. Event objects, simulation objects, and object managers can dispatch an event by scheduling a message.

Events (subclasses of C\_EVENT) encapsulate the data that is exchanged with simulation objects. Each event instance can only associate with a single simulation object instance, but a simulation object instance may have many events scheduled to act upon it. An event may test values on the corresponding simulation object instance via method calls and changes the instance's state by means of its method EXCHANGE.

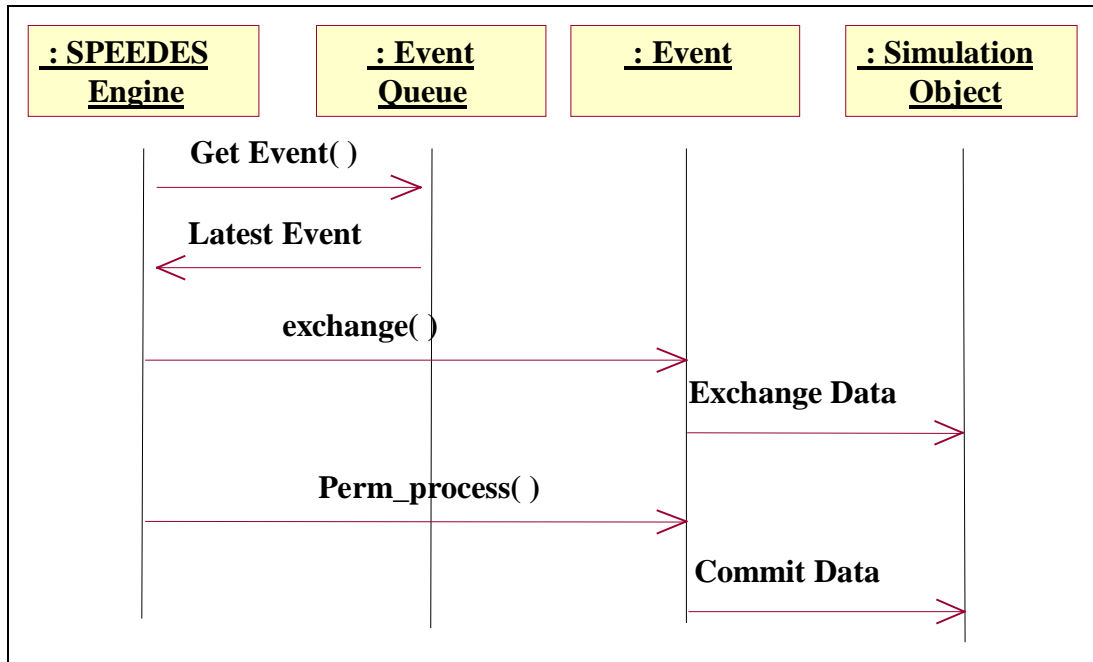
C\_EVENT also has the method PERM\_PROCESS, which commits the changes made by EXCHANGE. No rollbacks are processed after PERM\_PROCESS has been called. Commitment occurs when Global Virtual Time (GVT) is advanced. A GVT advancement is dispatched to all processing nodes in order to synchronize them. PERM\_PROCESS is also typically where data from events and simulation objects is output to external entities such as files, graphics, or standard output [14].

Fig. 9 shows an example of event creation once the simulation is executing. The object manager gets information from the simulation object and creates a message that is added to the internal event queue. This interaction between the object manager and the simulation engine is propagated through events.



**Figure 9 SPEEDES Event creation**

A typical event processing sequence is shown in Fig. 10. SPEEDES dispatches an event from the event queue to an available processor and calls the event's EXCHANGE method to swap event and object variable values. When global virtual time advances, PERM\_PROCESS is called and the data is committed on the simulation object.



**Figure 10 SPEEDES Event Execution**

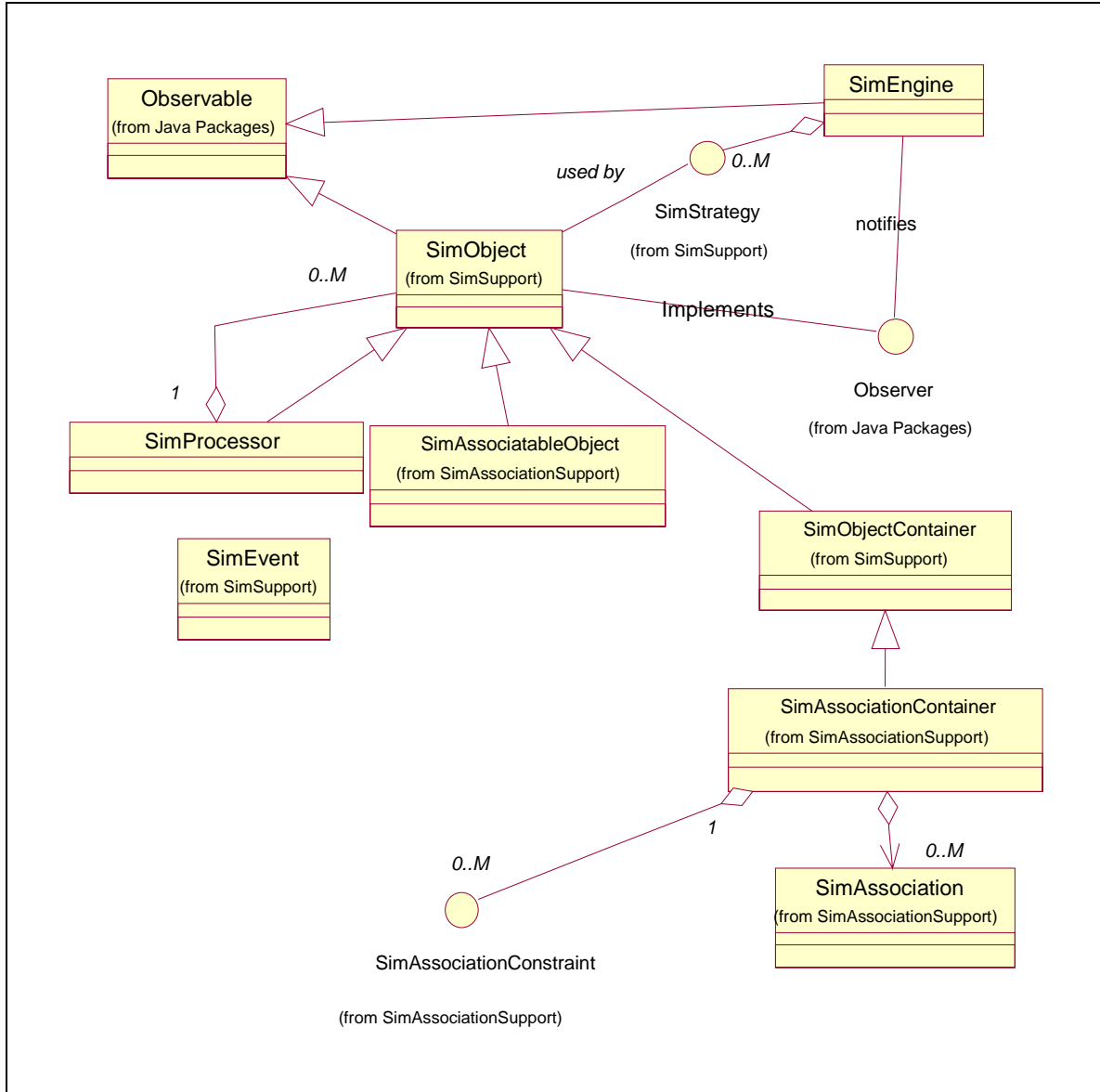
### 2.2.3 COMPOSE Compared to SPEEDES

Perhaps the most significant difference between COMPOSE and SPEEDES is the philosophy of the role of domain objects (simulation objects). In SPEEDES, the engine controls processing, and simulation objects are treated as data stores. Events are injected into the simulation via scheduled messages that contain the data to be exchanged with the simulation objects. Typically, the domain model is designed to be static and all activities are conducted within the interactions and constraints of events. The perspective of the domain is limited to a very event-centric view, while other views of the systems are excluded by the very nature of the SPEEDES package. COMPOSE, on the other hand, allows much more control over the domain objects. The main restriction of COMPOSE is that domain objects communicate via messages that are released into the COMPOSE engine via SEND\_MESSAGE. Domain objects are free to be activated via methods, which are tied to messages. In both SPEEDES and COMPOSE the large number of required events complicates the manipulation of associations.

## 3. AN ASSOCIATION ORIENTED SIMULATION PACKAGE: EASY

As with other simulation frameworks, EASY provides a set of superclasses that the simulation designer extends in order to create domain classes (Fig. 11). *SimObject* is the universal ancestor class and contains methods for handling events. *SimEngine* represents the simulation engine itself. *SimObjectContainer* manages and contains *SimObjects*. Associations are supported by *SimAssociationContainer*, *SimAssociatableObjects*, *SimAssociations*, and *SimAssociationConstraint*. Descendants of *SimEvent* represent the events used to communicate between ob-

jects. The following discussion of EASY focuses first on the event handling, then on the support for associations.



**Figure 11 Interactions of the simulation package**

### 3.1 Event handling

Events in EASY are designed such that much of the activity involved with association manipulation is encapsulated within the simulation package. There are two types of events:

1. Directed events where the target of the event is specified.
2. Notification events, which are forwarded to the appropriate destination by the simulation package.

Both directed and notification events can have a cascading property. A cascading event that is sent to an instance of a container class is automatically propagated to all the contained objects

The SimEvent class is based upon the command pattern [4], which allows requests to be encapsulated as objects. While SimEvent does not encapsulate an operation (except in the case of a constraint), it does cause activities within the simulation objects to take place.

Another important concept in EASY's event handling strategy is event propagation. EASY passes events up the inheritance chain to the correct level of abstraction where a handler for the event is found.

### 3.1.1 Directed Events

Directed events are targeted to specific SimObject instances. They are dispatched into the SimEngine and sent to the specific instance based on the event's timestamp. For example, a domain developer writes the following to create and dispatch a directed event called "reposition" to the instance myRadar1:

```
SimEvent repositionEvent = new SimEvent(myRadar1, "reposition", eventTime);
dispatchEvent(repositionEvent);
```

The event is created with a target instance (myRadar), the event type ("reposition") and a time (eventTime). At that time, the simulation engine dispatches the event by calling myRadar1's fireEvent method. Both myRadar1 and the object containing the above code must be instances of descendants of SimObject. SimObject has the methods dispatchEvent, which inserts an event into the simulation engine, and fireEvent, which accepts the event. Fig. 12 shows the dispatching of a directed event in the SimEngine.

```
public void dispatchEvent(SimEvent simEvent){
    if (simEvent.getSource() != null) { //directed event
        SimObject simObject=simEvent.getSource();
        simObject.fireEvent(simEvent);
    } else { //the notification event
        Pair thePair = null;
        for (int x=0; x< observers.size(); x++) {
            thePair = (Pair) (observers.elementAt(x));
            if (((String)(thePair.first)).equals(simEvent.getType())) {
                SimObject simObject=(SimObject)thePair.second;
                simObject.fireEvent(simEvent);
            }
        }
    }
}
```

Figure 12 Dispatching events in the SimEngine

### 3.1.2 Notification Events

Notification allows events to be broadcast to all listeners registered for an event type. A given simulation object can register for any number of notification events, and a single event can trigger reactions in any number of simulation objects. Any SimObject can register by calling its own registerForNotification method with the appropriate event type as a parameter. registerForNotification then calls the SimEngine's registerForNotification method. The SimEngine puts the SimObject and the eventName into a table sorted by eventName, as follows:

```
public void registerForNotification(SimObject so, String eventName) {
    Pair thePair = new Pair(eventName, so);
    observers.addElement(thePair);
}
```

The domain developer must add code to the SimObject's fireEvent method to react to each event. Internally, the SimEngine uses the Java Observable and Observer classes as a mechanism for informing instances of events. The dispatching of a notification event within the engine is shown in Fig. 12. A deregisterForNotification method with the same signature also exists in the SimEngine.

### 3.1.3 Cascading Events

A cascading event treats those SimObjects contained within a SimContainer as a single logical entity, reducing the number of events that the developer must create. A directed event or a notification event is cascading if it has the CascadeToContainedObjects flag set and is sent to a SimContainer object, which propagates a copy to each SimObject that it contains. The domain developer only needs to add code to handle events in the contained objects and call the setCascade method on the SimEvent. Fig. 13 shows an example of a fireEvent method where a cascading event is created.

```
public void fireEvent(SimEvent simEvent) {
    if ((simEvent.getType()).equals("Moved")) {
        SimEvent newSE = new SimEvent(RadarContainer.getInstance(),
            "constraint", simEvent.getTime());
        newSE.setCascade(true);
        dispatchEvent(newSE);
    }
}
```

**Figure 13** Creation of a cascading event.

### 3.1.4 Event Propagation

The Chain of responsibility pattern defines a succession of requests that are passed along a chain of handlers until one that can operate on each request is found or the event is ignored [4]. In a variation of this pattern, EASY uses inheritance as the chain of responsibility by taking

advantage of Java’s “super” construct. An event is passed up the inheritance chain until a handler for it is found. This is essentially an implementation of polymorphism. For example, if a Radar container instance receives an event that is not specific to that class, the event is handled by the SimObjectContainer superclass, which loops through the list of contained SimObjects and replicates the event for each contained instance. It is the responsibility of the modeler to include the call to super in the domain object.

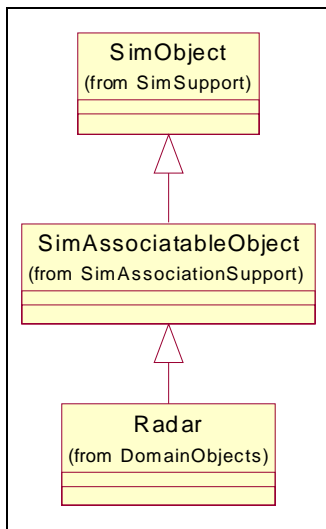
Fig. 14 shows the event handling code from the Radar domain class, and Fig. 15 shows the inheritance tree for a Radar. An event that is not of type “FOV” or “Pulse” is passed on to SimAssociatableObject.

```

public void fireEvent(SimEvent simEvent) {
    newTime = simEvent.getTime();
    String type = simEvent.getType();
    if (type.equals("FOV")) {
        changeFOV();
    } else if (type.equals("Pulse")) {
        togglePulse();
    } else {
        super.fireEvent(simEvent);
    }
}

```

**Figure 14 The Radar domain class handling events**



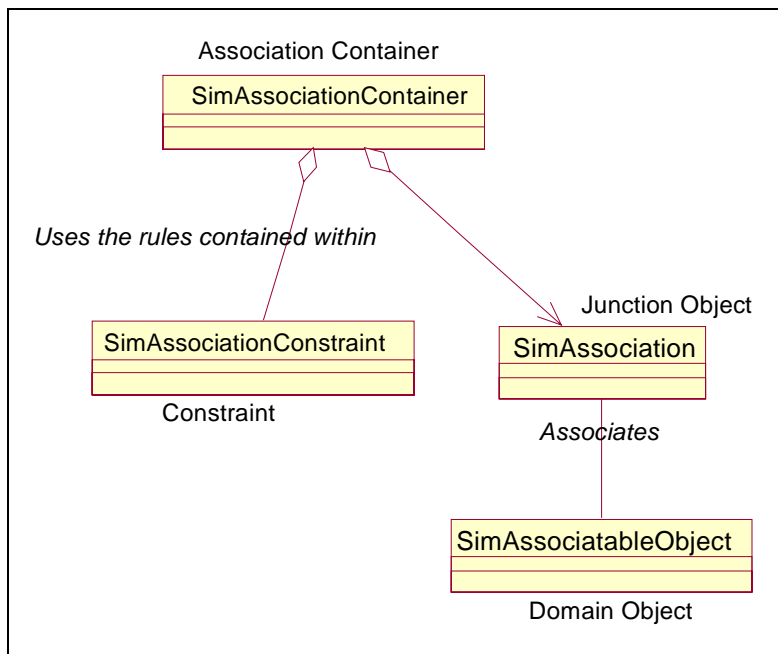
**Figure 15 Inheritance for a radar domain object**

### 3.2 Association support

Association support is based of the following five basic constructs:

- Domain objects (descending from SimAssociatableObject)
- Junction objects, representing linkages between the objects
- Association containers that contain various junction class instances
- Constraints to determine if the association should be formed
- Events that trigger the forming of an association

The corresponding classes (except events) are shown in Fig. 16. One or more concrete subclasses of SimAssociatableObject represent the domain objects. SimAssociatableObject contains methods for evaluating constraints related to the association and is a subclass of SimObject. SimAssociation contains a pointer array of SimAssociatableObjects and represents the junction between associated objects. A subclass of SimAssociationContainer is a container of SimAssociation instances. It is used for instance accounting and manipulation of SimAssociations. The SimAssociationContainer also contains the SimAssociationConstraint, which is used to evaluate SimAssociatableObjects and determine if an association should be created. SimEvents act as triggers for association manipulation.



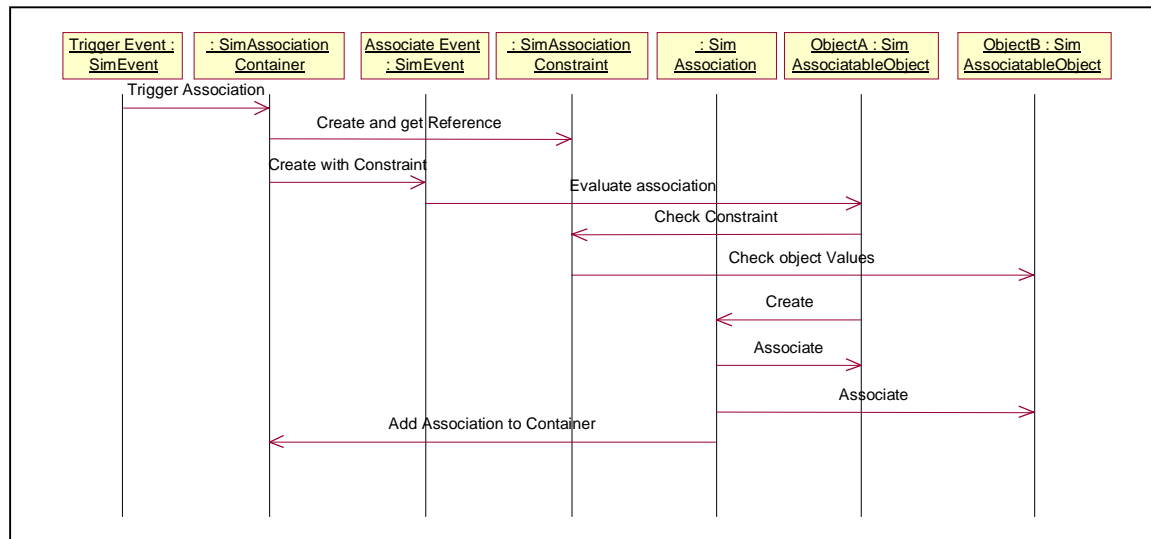
**Figure 16 Classes involved in the forming of an association**

Association formation is built on the mediator pattern [4]. This pattern encapsulates the interaction of objects and keeps the objects from explicitly referring to each other, which promotes loose coupling. It has three participants: an abstract mediator that defines the interface between colleagues, a concrete mediator (SimAssociationContainer) that implements the cooperation between colleagues (SimAssociatableObject), and several colleague classes that communicate via the concrete mediator.

SimAssociationConstraint contains the rules that determine whether an association should be formed. The concrete subclass of SimAssociatableObject that is evaluating the constraint passes the constraint class up the inheritance chain to SimAssociatableObject, which han-

dles the constraint. The chain of responsibility pattern is used extensively when forming associations, thereby delegating association work to the simulation package.

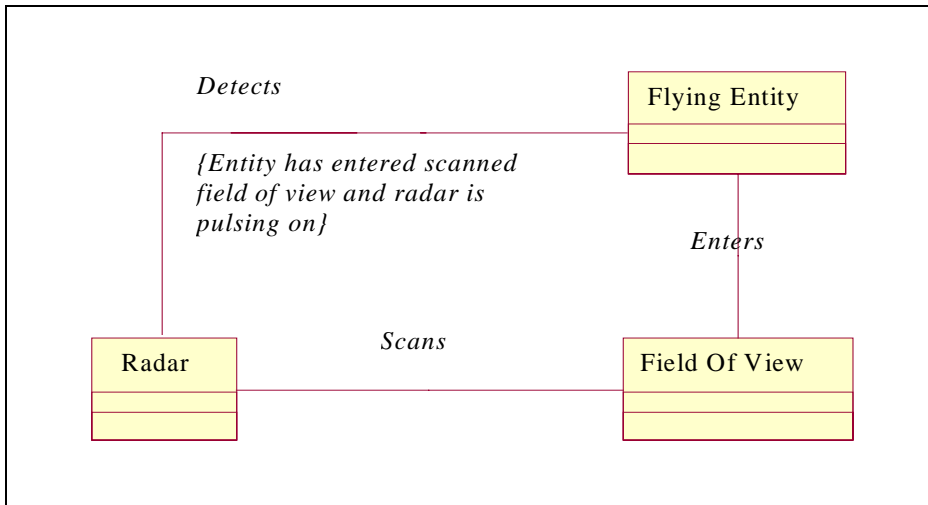
A typical event trace is shown in Fig. 17. The SimAssociationContainer listens for a triggering event. When it occurs, an event containing the SimAssociationConstraint is directed at one or more SimAssociatableObjects. The SimAssociatableObjects pass themselves to the SimAssociationConstraint for evaluation when they receive the constraint event. If the SimAssociationConstraint determines that the SimAssociatableObject's state is appropriate, a SimAssociation is created and added to the SimAssociationContainer.



**Figure 17 High-level trace of associating objects**

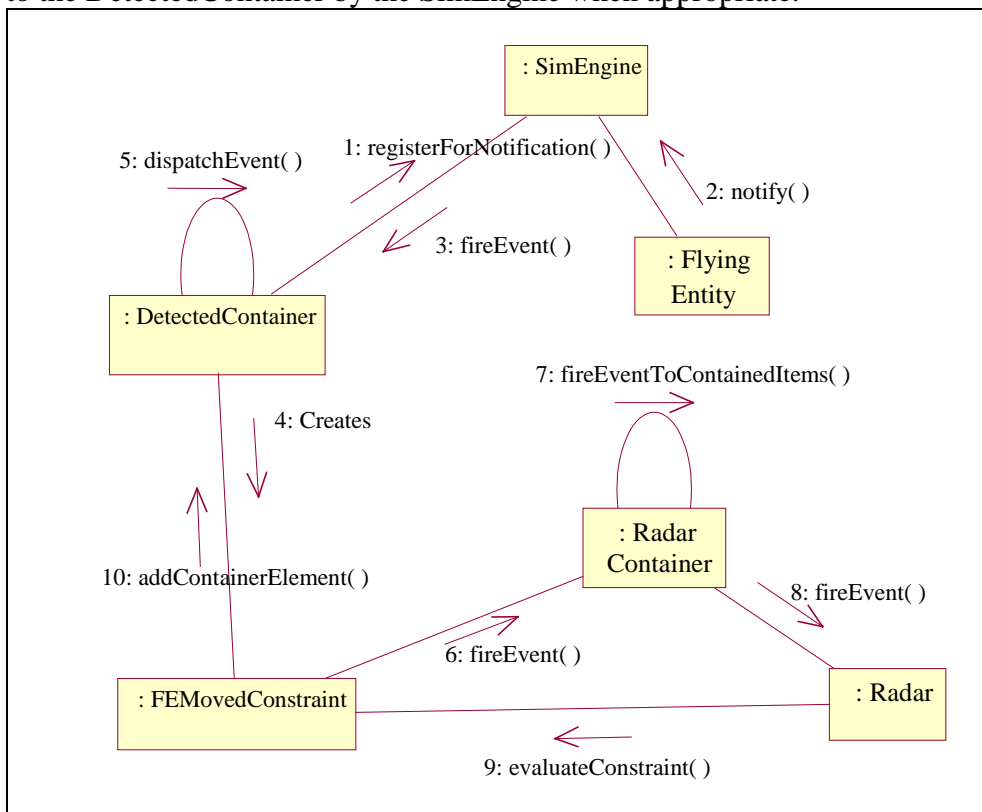
### 3.2.1 Example of Association Formation

The example domain includes a radar that periodically scans its field of view (FOV). A flying entity (FE) moving into the FOV is detected when the radar sends a detection pulse. The entity is no longer being detected when it moves out of the FOV. Fig. 18 shows the conceptual model. The detection association is formed when a flying entity enters the FOV and the radar sends a detection pulse.



**Figure 18 Conceptual model of radar/flying entities**

Fig. 19 shows a flying entity being detected by a radar, thereby causing an association to be formed. All objects except the SimEngine belong to domain classes. FlyingEntity and Radar are subclasses of SimAssociatableObject. DetectedContainer is a subclass of SimAssociationContainer. RadarContainer is a subclass of SimObjectContainer. FEMovedConstraint is a subclass of SimAssociationConstraint. Behind the scenes, the SimAssociation is formed and added to the DetectedContainer by the SimEngine when appropriate.



**Figure 19 Collaboration diagram of association being formed**

Here are the interactions shown in Fig. 19:

1. During construction, the DetectedContainer registers to be notified of "Moved" events as follows:

```
private DetectedContainer() {  
    super();  
    registerForNotification("Moved");  
}
```

2. During simulation execution, the flying entity generates a "Moved" notification event as follows.

```
SimEvent event = new SimEvent(null, "Moved", new Time);  
dispatchEvent(event);
```

3. The DetectedContainer instance is registered for this type of event so its fireEvent method is called with the "Moved" event as a parameter (Fig. 20). The method creates a constraint evaluation instance (FEMovedConstraint) and a new event of type "constraint" targeted toward the RadarContainer. The event is flagged to cascade to the contained radars.

4. An FEMovedConstraint object is constructed and added to the constraint event. The parameters for construction of the constraint are defined by the modeler.

5. The event is dispatched to the simulation engine.

```
public void fireEvent(SimEvent simEvent) {  
    if ((simEvent.getType().equals("Moved")) {  
        SimEvent newSE = new SimEvent(RadarContainer.getInstance(),  
            "constraint", simEvent.getTime());  
        newSE.setCascade(true);  
        FlyingEntity fe = (FlyingEntity) simEvent.getSource();  
        FEMovedConstraint movedConstraint =  
            new FEMovedConstraint(this, simEvent.getTime(), fe);  
        newSE.setConstraint(movedConstraint);  
        dispatchEvent(newSE);  
    }  
}
```

**Figure 20 A constraint notification is set up within the detectedContainer**

6. The "constraint" event is received by the RadarContainer, which does not know about it and passes it up the inheritance tree by means of the following call:

```

public void fireEvent(SimEvent simEvent){
    super.fireEvent(simEvent);
}

```

7. The event is eventually received by the SimObjectContainer. Since it has the cascade flag set it is dispatched to all contained instances via fireEventToContainedItems. The SimObjectContainer does not need to know what type of event it is; this is handled by the contained classes. Fig. 21 shows the fireEvent method of SimObjectContainer. The events are fired to each contained instance (namely Radars) via the fireEventToContainedItems method.

```

public void fireEvent(SimEvent simEvent) {
    SimObject simObject=simEvent.getSource();
    if (simEvent.cascade()) {
        fireEventToContainedItems(simEvent);
    } else {
        super.fireEvent(simEvent);
    }
}

protected void fireEventToContainedItems(SimEvent theEvent) {
    for (int x = 0; x< myVector.size();x++) {
        theEvent.setSource((SimObject)myVector.elementAt(x));
        dispatchEvent(theEvent);
    }
}

```

**Fig. 21 SimObjectContainer cascading events to contained classes**

8. When Radar receives a constraint, it passes it up the inheritance chain to SimAssociatableObject, which knows about constraints. Its fireEvent method is shown in Fig. 22.

```

public void fireEvent(SimEvent simEvent) {

    String type = simEvent.getType();
    if (type.equals("constraint")) {
        SimAssociationConstraint constraint = simEvent.getConstraint();
        constraint.evaluateConstraint(this);
    } else {
        super.fireEvent(simEvent);
    }
}

```

**Figure 22 SimAssociatableObject handles the constraint event**

9. The constraint is pulled from the SimEvent and evaluated in the evaluateConstraint method (Fig. 23). The Flying Entity, its X and Y coordinates, and the association container were passed to the constraint during construction. The method ensures that the entity is within the radar's field of view and that the radar's status is detecting.

The Radar class contains the evaluation methods used by the constraint. Fig. 24 shows the methods `getDetectingStatus`, which returns the state of the detection pulse, and `isInFOV`, which tests a point for being within the current field of view. The constraint object calls both evaluation methods whether the Radar is participating in a simulation or not. The Radar encapsulates its own state and the constraint encapsulates evaluation of the Radar state.

10. If the constraints are met, a new association is created and added to the `detectedContainer` instance.

```
public void evaluateConstraint(SimAssociatableObject theRadar){
    Radar radar = (Radar) theRadar;
    int x = flyingEntity.reportX();
    int y = flyingEntity.reportY();
    if ((radar.isInFOV(x,y)) && (radar.getDetectingStatus())) {
        DetectedAssociation detectedAssociation = new DetectedAssociation(time, radar,
                                                                    flyingEntity);
        simAssociationContainer.addAssociation(detectedAssociation);
    }
}
```

**Figure 23 FEMovedConstraint constraint evaluation**

```
public boolean getDetectingStatus() {
    return detectionFlag;
}

public boolean isInFOV(int inX, int inY) {
    return ((inX.isIn(FOV) && (inY.isIn(FOV)));
}
```

**Figure 24 Constraint evaluation methods**

This example shows how EASY allows domain objects to be implemented with few implementation level constructs and loose coupling with regards to associations. Most of the work is handled in classes provided by the simulation package. The association container listens for a notification. The constraint object is tied directly to the association container class. No domain object is directly involved in handling the constraint evaluation, but merely provides methods for the constraint to call. What this means is that any number of constraints could be added without affecting the domain classes.

From the domain objects' point of view, the association formation is as follows, with references back to the steps indicated in the collaboration diagram in Fig. 19:

The Flying Entity object notifies the simulation of a Moved event. (Step 2.) This is a notification event so the flying entity does not have to know what the recipients are or why they need to be notified.

The evaluateConstraint method of FEMovedConstraint is called with a Radar object as a parameter. (Step 9).

The FEMovedConstraint object registers the association with the association container. (Step 10).

The domain objects are not coupled to each other. They pass notification messages and constraint evaluation to the simulation package, which handles the operation by dispatching the events to the correct simulation objects. Within the simulation package, container classes handle broadcasts to contained instances, and associatable objects handle constraint evaluation.

## 4. CONCLUSION

EASY reduces the artificial constructs necessary for forming and dissolving associations within the domain model. The abstractions added by EASY foster an undistorted and conceptually clean domain model. Domain objects are coupled via the simulation engine and not directly to each other, fostering good cohesion and loose coupling. Modeling a constraint as a class leads to an implementation that easily traces back to the conceptual model.

The example shows that the operations for associations are clustered at a good level of abstraction. The flying entity and radar domain objects are involved in the association only from the standpoint that they interact with the simulation. The complexity of creating associations and broadcasting events via notification and cascading to contained objects is handled by the simulation package.

EASY has been shown to induce less complexity into a domain model than a process oriented or a state manipulation simulation package [15]. A set of object-oriented complexity metrics were applied to a sample domain model implemented using representative simulation frameworks. EASY's metrics indicated a less complex implementation than comparable simulation packages. Mechanisms such as event propagation, cascading event, and association constructs, were shown to be keys to reducing the complexity of handling associations. These techniques can be used as a basis for building an abstraction layer for many different simulation packages.

## References

- [1] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- [2] Fishwick, P. An integrated approach to system modeling using a synthesis of artificial intelligence, software engineering and simulation methodologies, *ACM Transactions on Modeling and Computer Simulation* 2, no. 4 (1992): 307-330.
- [3] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

- [4] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.
- [5] Gottlob, G., Schrefl, M., and Rock, B. Extending Object-Oriented Systems with Roles, *ACM Transaction on Information Systems* 14, no. 3 (1996): 268-296.
- [6] Martin, J., Bagrodia, R. COMPOSE: an Object-Oriented Environment for Parallel Discrete-Event Simulations. Proc. Winter Simulation Conference 1995.
- [7] Martin, D., McBrayer, T., Wilsey, P. Warped: A Time Warp Simulation Kernel for Analysis and Application Development, Proc. 1996 Hawaii International Conference on System Sciences
- [8] Odell, J. , Fowler., M. From Analysis to Design Using Templates. *Report on Analysis and Design*, March 1995.
- [9] Overstreet, C., Nance, R. World View Based Discrete Event Model Simulation. In *Modeling in the Artificial Intelligence Era*, 165-179: North Holland, 1986.
- [10] Page, E., and Nance, R. Parallel Discrete Event Simulation: A Modeling Methodological Perspective, *ACM SIGSIM Simulation Digest* 24, no. 1 (1994): 88-93.
- [11] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [12] Steinman, J. SPEEDES: synchronous Parallel Environment for Emulation and Discrete Event Simulation, Proc. Advances in Parallel and Distributed Simulation 1991, 95-103
- [13] Steinman, J. Breathing Time Warp. Proc. 7th Workshop on Parallel and Distributed Simulation (PADS '93), 1993.
- [14] Steinman, J. Discrete-Event Simulation and the Event Horizon. Proc. 1994 Parallel And Distributed Simulation Conference (PADS '94), 1994.
- [15] Suscheck, C. EASY: An association oriented simulation framework, D. CS. Dissertation, Colorado Technical University, 2000.
- [16] Whitehurst, R. Alan. Association Frameworks in Simulation Reuse, Computer Simulation Society Proceedings on Object-Oriented Simulation, 1997.