Towards a Distributed Object-Oriented Propagation Model Using Ada95

Donald M. Needham
Computer Science Department
United States Naval Academy
Annapolis, MD, USA 21401
+1 410 293-6812
needham@scs.usna.navy.mil

Steven A. Demurjian, Sr.
Computer Science & Eng. Dept.
The University of Connecticut
Storrs, CT, USA 06269
+1 860 486-4818
steve@eng2.uconn.edu

Thomas J. Peters
Computer Science & Eng. Dept.
The University of Connecticut
Storrs, CT, USA 06269
+1 860 486-4818
tpeters@eng2.uconn.edu

ABSTRACT
Representing interdependencies between the objects of an object-oriented software application requires design-time mechanisms for specifying object interrelationships, as well as software constructs for the runtime maintenance of these relationships. In this paper, we present a portion of our software engineering research environment ADAM, (short for Active Design and Analyses Modeling), which incorporates a technique for the design-time modeling of *propagations* (our term for the relationships between interdependent objects). We examine the ADAM environment's support for the automated generation of Ada95 software constructs that maintain object interdependency at runtime. We focus on our propagation model's use of Ada95 tasking constructs and protected objects, with an emphasis on the source level mechanisms through which our model utilizes concurrency. We present constructs required for an Ada95 distributed propagation model that supports communication through CORBA.

Keywords
Object-oriented technology, Methods and Techniques, Ada Language, Distributed, CORBA.

1 INTRODUCTION

Representing dynamic (runtime) interdependencies between objects is an essential part of using object-oriented techniques to model the critical software communications found in today's increasingly complex software systems. Current software design methodologies, software development environments, and computer aided design (CAD) systems lack the ability to represent and manipulate these interdependencies, even though the specific information that must be represented is often well understood by the designer. Representing dynamic interdependencies in an object-oriented design environment focuses on the ability to specify relationships between objects that are not related ancestrally. For example, a CAD designer altering the pitch of a gas turbine engine's fan blade design may need to consider the stress where the blade is mounted to the turbine disk. Changes in the stress (caused by the change in fan blade pitch) are said to *propagate* to the attachment of the fan blade to the disk. *Propagation modeling* seeks to represent such interdependency knowledge as an important part of the design process.

To facilitate our discussion of propagation modeling, we define the following concepts, and briefly examine a propagation found in the gas turbine engine fan blade design domain [9]. We define a *propagation* to be a software construct for specifying relations between two or more non-ancestrally related objects. Each propagation is an independent, third-party object that coordinates the interdependencies between all of the objects in the propagation.

Objects are interrelated if they need to exchange messages, via the propagation, in order to maintain their collective state information consistent with their interdependencies. Two distinct categories exist into which all of the interrelated object types of a propagation are partitioned:

1. *Source objects*: The interrelated objects that collectively determine the conditions under which the propagation's control-logic is invoked.
2. *Destination objects*: The interrelated objects that do not participate in determining when a propagation's control-logic is invoked.

Propagation *resolution* is the final activity of a propagation, and involves informing the object(s) invoking the propagation as to whether or not the propagation was successful. *Propagation-required methods* are those methods that are subject to invocation during the course of a propagation's execution.

Figure 1 shows a propagation model for the pitch alteration of a gas turbine engine fan blade [9]. In Figure 1, the boxes represent interrelated object types, and

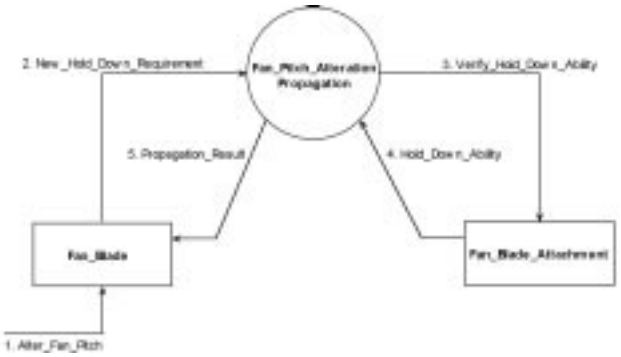

Figure 1: Propagation For Fan Blade Pitch Alteration.

the circle represents the propagation entity. The sequentially labeled arrows represent the communication, overseen by the propagation, needed to service the interdependencies between the interrelated objects. As shown in Figure 1, propagation execution proceeds as follows:

1. The propagation is initiated by the source object instance (`Fan_Blade`) being requested to fulfill its `Alter_Fan_Pitch` responsibility. In our example, the propagation designer has determined that the `Alter_Fan_Pitch` responsibility of the `Fan_Blade` instance is interrelated with a destination object, in this case the `Fan_Blade_Attachment` instance.

2. The source object responds by launching the `Fan_Pitch_Alteration` propagation, passing information needed by the propagation via the automatically dispatched `New_Hold_Down_Requirement` message.

3. The `Fan_Pitch_Alteration` propagation coordinates the activity of the interrelated objects, determining whether the `Fan_Blade_Attachment` has adequate hold down capabilities in light of the *proposed* new pitch of the `Fan_Blade` instance.

4. The `Fan_Blade_Attachment` instance determines and reports its own ability to hold down the fan blade (via the `Hold_Down_Ability` message).

5. Finally, the propagation communicates the result of the propagation to the source object via the `Propagation_Result` message sent to `Fan_Blade` instance.

The goals of the communications described in this example are to insure that if the application state was consistent before propagation launching, then the application state remains consistent after propagation resolution. In terms of our example, if the fan blade attachment could sustain the stress generated by the fan blade with its initial pitch, then the propagation asserts that the case with the altered fan blade also meets the stress requirements. In order to provide such assurances, information consistency requires that a propagation terminate after either ensuring that appropriate alterations have been made to the interrelated objects (propagation success) through the use of the propagation-required methods, or that no alterations have been made (propagation failure). After propagation resolution, the source object resumes execution of its original responsibility, perhaps making use of the resolution result returned by the propagation.

The remainder of this paper is organized as follows. In Section 2 we examine our approach to propagation modeling, with an emphasis on the phases of a propagation. In Section 3 we briefly present the ADAM environment, with a focus on the concept of profiling and ADAM's support for integrity constraints. In Section 4 we present the propagation-specific code generation capabilities of the ADAM environment, including our use of the Ada95 tasking model to achieve concurrency within our propagation modeling implementation. In Section 5 we assess our propagation model's impact on specifying object interdependency during the design process. In Section 6 we examine recent work that has been done in areas related to propagation modeling. In Section 7, we present our conclusions, and discuss directions for future research.

2 MODELING PROPAGATION

Our object-oriented propagation model focuses on allowing a designer to specify interdependencies between design objects. The object-oriented concept of message passing is relied upon as the primary means of communication between a propagation's interrelated objects, and the propagation constructs we have developed. We build upon objects with integrity constraints [5], which allows us to focus on the interdependent activities of objects without concern for an object's specific internal behavior. Our approach allows the internal behavior of a propagation's interrelated objects to be modified as needed during the iterative design process, as an event distinctly separate from the process of specifying a propagation.

We mandate a protocol of propagation-required behavior so as to standardize the communication between a propagation and its interrelated objects. The particular propagation-required behavior for an object depends upon the object's role in the propagation. The number of source and destination objects determine the *configuration* of a propagation. All of the source objects of a propagation are endowed with a fixed set of propagation-required behavior, while the destination objects are given a subset of a source object's propagation-specific behavior requirements.

To aid the designer in specifying the interdependencies between objects, our propagation model provides constructs for both sequential and conditional message passing. The sequential message constructs allow a designer to identify the serial ordering of method invocation between a propagation's interrelated objects. In the gas turbine engine fan blade example of Figure 1, the messages that are exchanged between the `Fan_Blade` object and the `Fan_Blade_Attachment` object while the pitch of the `Fan_Blade` object is being altered are strictly sequential in their logical flow. The other basic constructs, the conditional message constructs, allow a designer to specify sequences of message passing that are dependent upon response to queries made dynamically by the propagation construct. An example of a conditional message sequence would be the fan blade example modified such that the *length* of the fan blade was the subject of the propagation, rather than the pitch. When the designer specifies a change in the length of a fan blade, the propagation entity might be designed so as to allow an altered fan blade length only if the new fan blade length does not exceed some maximal distance, as determined by the radius of the engine casing, while at the same time the altered stress placed on the fan blade Attachment by the increase in length is within the limits specified by the `Fan_Blade` object. Due to space constraints, we limit our examination of propagation constructs to those supporting sequential message passing, and refer the interested reader to a more complex example from the manufacturing domain [9].

As part of our abstraction efforts, we encapsulate the message passing, overseen by the propagation in servicing the interdependencies between its interrelated objects, into a trigger[1] method. This approach allows the designer to focus on specifying only those interactions between objects that are necessary to maintain *application-wide* information consistency, while the reusable portions of our model provide the underlying propagation message passing that supports the designer's trigger method definition.

The Role of Integrity Constraints

Changes to the information maintained within an object-oriented software application result primarily from the messages passed between the application's objects. The role of propagation modeling is to enforce information consistency across the entire application, a level of abstraction higher than that of the object-type level. To achieve application-wide information consistency, propagation modeling requires special behavior from the objects interrelated in a propagation. We utilize objects endowed with class-level assertions, which we term integrity constraints (IC)s [5] (further discussed in Section 3). Our specific requirements are that interrelated objects:

1. must maintain their own information consistency (via their internal ICs),
2. must not alter their state information upon receipt of an IC-violating[2] message, and
3. must report the receipt of any such IC-violating message to the governing propagation entity.

The first and second capabilities allow a propagation to deal abstractly with interrelated object types (OTs), with the knowledge that the OTs only alter their state information in response to messages consistent with the respective OTs internally maintained ICs. The third capability, reporting the receipt of IC-violating messages, is used during the course of a propagation to guide the behavior of the propagation entity, and allows the propagation entity to alter its behavior in response to the effect it has had on the interrelated OTs.

The Basis for a Propagation Type

To facilitate information consistency across the entire application, we utilize a *propagation type (PT)* modeling construct [9]. A PT is a mechanism for allowing a software designer to specify IC-like behavior that occurs across multiple OTs. The PT provides the vehicle through which the designer specifies the interdependencies between interrelated OTs, and identifies the source OTs that are consulted in the determination of when to service these interdependencies. Much like the development of the OTs of an application, the development of a PT is an iterative process wherein a propagation is refined and redefined as additions/changes are made to the application.

The role of an instance of a PT, referred to as a *propagation entity (PE)*, within the runtime environment of a software application includes:

- Enforcing the dynamic behavior, defined via the PT at design time, of the runtime application.
- Servicing the interdependencies between relevant portions of an application's data.

[1] We build upon the concept of *triggering* from database research [8] wherein the enforcement of database constraints relies upon implementation-level triggers. Database triggers automatically take restorative action upon determining that a database constraint has been violated. In our propagation model, we use the term triggering to reference the events that occur when a propagation's control-logic is invoked to enforce the interdependent behavior of the source and destination objects.

[2] An IC-violating message is a message received by an object that requests state changes incompatible with the object's specified integrity constraints.

- Notifying the object activating the PE in the case that the requested changes are not permissible in terms of the information consistency of the propagation's interrelated objects.

A many-to-one relationship exists between PEs and PTs. Each PT can have multiple PEs, activated as needed by objects during the application's runtime. Each PE is *associated* with the appropriate OT instances at runtime, and interacts with its interrelated objects via message-passing. It is important to note that a PE is itself an object instance, and follows the same scoping and lifetime requirements as the objects whose interrelationships the PE oversees.

A PE has three distinct phases that it may shift between during its execution. These phases determine the internal activity of the PE. A PE changes between phases explicitly, generally as a result of a message passed to the PE from one of its interrelated objects. The phases of a PE are given as:

1. **Propagation Initialization:** The propagation initialization phase is a short-duration, transitory phase in which the PE is associated with the interrelated objects. A unique ID is assigned to each PE, allowing a source object to distinguish amongst its association with one PE or another. Unique IDs also permit the design of applications that require more than one PE of a given PT to be active at any one time.
2. **Propagation Waiting:** After the initialization phase, the PE automatically enters the propagation waiting phase. In this long-duration phase, the PE responds to messages from the associated source objects. In particular, a source object sends source-ready messages indicating that the source object requests that the propagation be triggered. Upon receipt of a source-ready message, the PE updates its state table and, depending on the modified contents of the state table, determines whether the PE should change to the propagation triggering phase.
3. **Propagation Triggering:** The propagation triggering phase is another short-duration, transitory phase in which the PE attempts to service the interdependencies between its objects. After invoking the interdependency control-logic, the PE automatically returns to the propagation waiting phase.

In servicing the interdependencies between the designated portions of the application's data, our propagation model requires a PE to have the ability to restore its interrelated objects to their pre-propagation states. It is our view that a PE's triggering events should not proceed further if, during an interdependency-related message passing sequence, an interrelated object reports the receipt of the IC-violating message. In this case, although the particular OT reporting the receipt of a IC-violating message is internally consistent, the state of the application when viewed from an inter-object perspective may have become inconsistent. Our approach to the PE's receipt of an IC-violating message during the course of propagation triggering is to:

1. require the propagation model to halt the PE's trigger method's message passing sequence,
2. restore all of the interrelated objects to their respective pre-propagation states, and
3. communicate to the activator of the PE that the propagation failed due to an interrelated object's receipt of an IC-violating message.

A PE considers entering its triggering phase only in response to the receipt of a source-ready message from one (or more) of its source objects, since the receipt of a source-ready message is the only event that can cause a PE's state to be modified in a manner that would indicate that the PE should trigger. After the receipt of a source-ready message, a PE can be queried as to its status. The status of a propagation includes whether the propagation was successful, failed, or is still waiting for collective agreement from the source OT instances over whether to trigger. In the case of a failed propagation, which occurs when an interrelated OT instance receives an IC-violating message, the PE's status will also include the identity of the OT instance reporting the IC violation. Such querying provides the application with detailed information regarding the status of a propagation. In support of such querying, each PE maintains state information on its *most recent* response to a source-ready message. We identify the possible propagation states as:

1. `Prop_Waiting`: The PE has not entered its triggering phase since being initially created or reset.
2. `Prop_Success`: The PE has successfully completed its triggering phase, and has serviced the interdependencies between its objects without any object reporting the receipt of an IC-violating message.
3. `Prop_Failure`: The PE has unsuccessfully completed its triggering phase due to an interrelated object having reported the receipt of an IC-violating message. The states of all of the interrelated objects are identical to their pre-propagation states.

Upon receipt of a message from a source object indicating the object's willingness to participate, the PE determines whether or not to enter its transitory triggering phase. This determination is made through the PE's inspection of its source-ready state table. If the PE does enter the triggering phase, the PE reports either prop_success or prop_failure depending on whether any IC-violating messages were reported by the interrelated objects during the PE's triggering phase. Otherwise, if the PE does not elect to enter its triggering phase, the PE reports that it has remained in the waiting phase.

The Role of Objects in Propagation

In Section 1 we defined the source and destination OTs of a propagation. We now more thoroughly examine these categories. The *source OTs* of a propagation are those OTs that collectively determine the conditions under which a propagation enters its triggering phase. Our view of the source objects is that the propagation only triggers if all of the source objects are in agreement as to whether the propagation should fire. Towards this end, each source object communicates, to its associated PE, the source object's readiness to participate in the propagation through the following messages:

1. `Source_Ready`: The source object is ready to participate in the propagation.
2. `Source_Unready`: The source object is not ready to participate in the propagation.
3. `Source_Failure`: The source object will not become ready to participate in the propagation.

The `Source_Ready` message is used by a source object to positively indicate the object's readiness for the propagation control-logic to be invoked. The PE's receipt of a `Source_Ready` message does not necessarily mean that the propagation control-logic will be invoked, but rather causes the PE to analyze, given the updated readiness of the source object passing the ready message, whether all of the sources are now ready for the PE to enter the triggering phase. The `Source_Unready` message is used by the source object to indicate to an associated PE that the source object is not ready to participate in the propagation. The PE, in response to the receipt of a `Source_Unready` message, updates its state table appropriately. The `Source_Failure` message provides a mechanism through which the PE may fail without ever having attempted to service the interdependencies between its objects. Because source OTs may be defined as participating in any number of PEs, source objects additionally require the ability to identify the PEs with which they have been associated.

Destination objects do not play a role in determining whether the propagation enters its triggering phase, and participate in the propagation only to the extent that their states are subject to alteration during the PE's triggering phase. OTs may participate as sources in some PTs and/or as destinations in other PTs.

3 PROPAGATION CODE GENERATION

In this section, we briefly present the software engineering research environment in which our propagation model is embedded. We then use our model to develop the previously discussed propagation found in the gas turbine engine fan blade example. Finally, we examine the propagation-specific `Ada 95` source code generated by our modeling environment.

A Context For Propagation Modeling

ADAM (short for **A**ctive **D**esign and **A**nalyses **M**odeling) is an object-oriented design model and environment that is utilized both as a research testbed and for teaching undergraduate software engineering at The University of Connecticut [2]. ADAM supports an object-oriented design model that is tightly integrated into the environment, with the semantic scope, content, and context of each modeling construct clearly defined. There are no specific programming language dependencies in ADAM; design choices are made via menus, browsers, etc., and text is directly entered by the designer using forms and browsers. The environment stresses language independence by focusing on design and allowing code to be automatically generated in a variety of target languages, including, `Ada 83`, `Ada 95`, `GNU C++`, `Ontos C++`, a dialect of `Common LISP` and `Eiffel`. ADAM supports an incremental and iterative approach to the design process by allowing design data to be stored persistently in the Ontos object-oriented database system.

One of the key concepts and constructs of ADAM is the *profile* [2]. This modeling construct tracks the purpose and intent of the different design choices and components in an application. Profiling occurs throughout the design and development process to promote the understandability of the solution, to insure its compliance with the requirements, and to provide feedback and guidance in its refinement.

An ADAM modeling of the object types (OTs) used in the gas turbine fan blade application appears in Figure 2. In Figure 2, five object types appear: the `Blade` OT, which represents an abstraction for various types of blades, the `Fan_Blade` OT, which inherits from the `Blade` OT, the `Connector` OT, an abstraction for various fasteners, and the `Fan_Blade_Attachment` and `Z_Fitting` OTs, both of which inherit from the `Connector` OT.

The ADAM environment's integrity constraint (IC) construct [5] spans both programming (typing) and database (derived value) requirements for integrity in

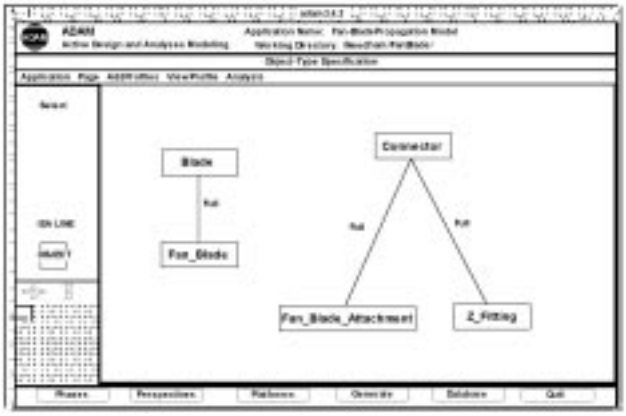

Figure 2: Fan Blade Object Types.

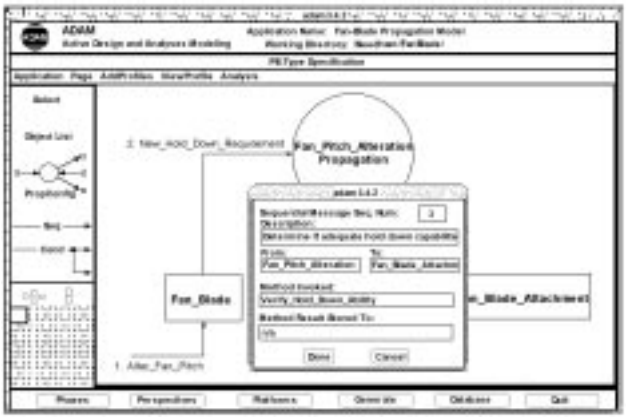

Figure 3: The Sequential Message Profile.

an object-oriented design model. The ADAM environment takes the view that ICs are restricted to act exclusively within an OT and define the values that an attribute may take. An IC applies to a single instance of an OT, hence the behavior of an IC is encapsulated within a single object. A constraint may not involve the private data items of two separate instances even if the two instances are of the same OT.

ADAM utilizes the information provided during the design process via profiles to create implementation classes that correspond to user-defined object types. Although ADAM supports the generation of source code in multiple languages, this paper focuses on the generation of `Ada95` source code. When `Ada95` is selected as ADAM's targeted generation language, a type encapsulated within a package is produced for each user-defined object type, and both a protected object and a task type is produced for each propagation type (PT).

We use the design of the gas turbine fan pitch alteration propagation described in Figure 1 to examine message passing in propagations. The fan pitch propagation involves a single source OT, `Fan_Blade` and a single destination OT, `Fan_Blade_Attachment`, both of which appear as leaf nodes in Figure 2. We now turn our attention towards the process through which the designer of the Fan_Pitch_Alteration propagation embeds the desired propagation control-logic for use in the *triggering* phase. The first message (`Alter_Fan_Pitch`) is an external request made of the Fan_Blade OT to alter its fan pitch, which in turn launches the propagation. The second message, `New_Hold_Down_Requirement`, is passed from the `Fan_Blade` object to the `Fan_Pitch_Alteration` PT. In Figure 3, the designer is in the process of adding the third message, `Verify_Hold_Down_Ability`, from the `Fan_Pitch_Alteration` PT to the `Fan_Blade Attachment` OT.

The foreground of Figure 3 gives the ADAM Sequential Message Profile dialog box. The Sequential Message Profile allows the propagation designer to enter the sequential order of the message being passed between the propagation and the interrelated object. In Figure 3, the designer has indicated that the third message in the propagation sequence is `Verify_Hold_Down_Ability`, which is passed from the `Fan_Pitch_Alteration` PT to the `Fan_Blade_Attachment` OT. Figure 4 shows the final ADAM screen after the `Fan_Pitch_Alteration` propagation has been fully modeled with all five of the required sequential messages.

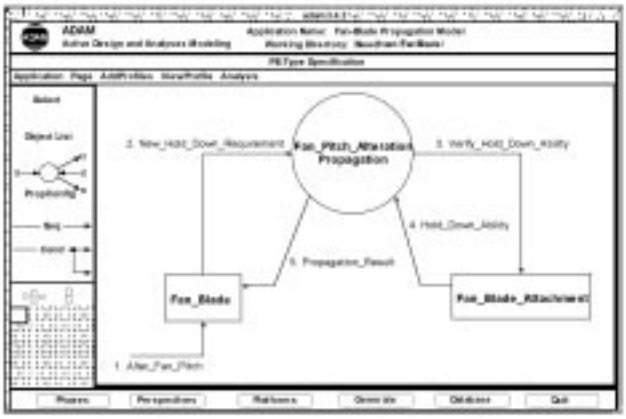

Figure 4: Completed Fan_Pitch_Alteration Prop Model.

4 ADA95 PROPAGATION CONSTRUCTS

In this section, we examine the propagation-specific Ada95 source code automatically generated by the ADAM environment. ADAM's automatic source code generation capabilities rely upon the information communicated to ADAM by the designer through the ADAM profiles used to create Figures 2, 3, and 4. In this paper, we limit our source code discussion to the task and protected object Ada95 constructs that implement behavior common to all propagations

```
1  TASK TYPE FanPitchTaskType IS
2      ENTRY Associate(FanBlade_in    : IN OUT FanBladeType;
3                      FanBlAttach_in : IN OUT FanBlAttachType;
4                      PropIdDameon   : IN OUT PropIdDameonType);
5      ENTRY SrcReady (FanBlade_in    : IN OUT FanBladeType;
6                      FanBlAttach_in : IN OUT FanBlAttachType;
7                      PropResult     : IN OUT PropResultType);
8  END FanPitchTaskType;
```

Figure 5: Fan Pitch Prop Task Specification.

Our propagation model supports concurrency in the runtime servicing of an application's propagation-related events. The ADAM environment supports such concurrency by generating Ada95 task constructs within which the behavior specific to each propagation is encapsulated. Figure 5 gives the specification for the task type produced by ADAM for our fan pitch alteration propagation example. As shown in Figure 5, each task has an `Associate` and `SrcReady` entry. The `Associate` entry is used by the propagation entity as part of the propagation *initialization* phase described in Section 2, while the `SrcReady` entry supports a propagation's *waiting* phase and *triggering* phase.

Figure 6 gives the body of the `FanPitchTaskType`. The `Trigger` operation (lines 4-13 of Figure 6) constitutes a major portion of a propagation's *triggering* phase. The `Trigger` operation houses the specific communication required between the source and destination objects (here the `FanBlade` and the `FanBladeAttachment` instances), as indicated by the designer via the ADAM environment (Figure 4). Note that each operation invoked in lines 9-12 of the `Trigger` operation raises a `PropException` if the operation requests that the object being acted upon would violate an integrity constraint by fulfilling the operation.

Upon activation of a fan blade propagation task [1], as part of the *initialization* phase, the propagation waits for the required association between a fan blade object and a fan blade attachment object. Lines 15-20 of Figure 6 give the definition of the `Associate` task entry, which requests a unique identifier for the propagation thread and then passes the identifier to each source instance (in this case the `FanBlade` object). Lines 21-38 of Figure 6 show the general behavior of a propagation. The propagation thread waits, at the open select alternative in line 23, for a `SrcReady` rendezvous. The `SrcReady` entry is used by the interrelated objects to indicate that the source object of the propagation thread is ready for invocation of the the interrelated behavior between the source and destination objects. Once the propagation thread accepts the `SrcReady` rendezvous and enters the *triggering* phase, the propagation validates that the source object is same source object with which the propagation has been associated. Deep copies

```
1  TASK BODY FanPitchTaskType IS
4    PROCEDURE Trigger(FanBlade    : IN OUT FanBladeType;
5                      FanBlAttach : IN OUT FanBlAttachType) IS
6        HoldDownReq : HoldDownReqType;
7        HoldDownAbilityResult : HoldDownAbilityResultType;
8      BEGIN  -- Trigger
9         HoldDownReq := NewHoldDownReq(FanBlade);
10        VerifyHoldDownAbility(FanBladeAttach, HoldDownReq);
11        HoldDownAbilityResult := HoldDownAbility(FanBladeAttach);
12        PropResult(FanBlade, HoldDownAbilityResult);
13     END Trigger;
14 BEGIN -- Task Body FanPitchTaskType
15   ACCEPT Associate(FanBlade_in    : IN OUT FanBladeType;
16                    FanBlAttach_in : IN OUT FanBlAttachType;
17                    PropIdDameon   : IN OUT PropIdDameonType) DO
18       PropIdDameon.GetUniquePropID(UniquePropID);
19       SetSrcIdent(FanBlade_in, UniquePropID, SRC1);
20   END Associate;
21   LOOP
22    SELECT
23      ACCEPT SrcReady(FanBlade_in    : IN OUT FanBladeType;
24                      FanBlAttach_in : IN OUT FanBlAttachType;
25                      PropResult     : IN OUT PropResultType) DO
26          Validate(FanBlade_in);
27          ... copy source/destination for possible restoration
28          Trigger(FanBlade_in, FanBlAttach_in);
29          PropResult := Success;
30            EXCEPTION
31             WHEN PropException =>
32                ... restore original source and destination
33               PropResult := Failure;
34       END SrcReady;
35    OR
36       terminate;
37    END SELECT;
38   END LOOP;
39 END FanPitchTaskType;
```

Figure 6: Fan Pitch Prop Task Body.

of the the source and destination objects are then made for use in the event that the `Trigger` operation requests IC-violating behavior of any of the interrelated objects. The `Trigger` operation is then attempted, which results in the propagation thread returning a success indication to the invoker (line 29) if the interrelationships desired by the designer have been maintained. Conversely, if IC-violating behavior is attempted during the `Trigger` invocation, the propagation thread (lines 30-33) restores the source and destination objects to their pre-propagation states, and a propagation failure message is sent to the invoker of the propagation. The propagation thread then re-enters the *waiting* phase.

Figure 7 shows the declaration for the protected object type `PropIdDameonType`. The `PropIdDameonType` allows our propagation model to provide a unique identity to each propagation created. A unique identity is required by each propagation instance so that each propagation can distinguish the specific object instances with which the propagation is associated. The `PropIdDameonType` is a protected object, with a protected procedure `GetUniquePropID`. The protected procedure `GetUniquePropID` (line 2 of Figure 7) provides exclusive read-write access to the data `UniquePropIDStore` of the protected object. Our use of

```
1 PROTECTED TYPE PropIdDameonType IS
2      PROCEDURE GetUniquePropID (NewID : OUT PropIDType);
3      PRIVATE
4          UniquePropIDStore : PropIDType := PropIDType'First;
5 END PropIdDameonType;
```

Figure 7: Prop Ident Dameon Protected Object Type.

a *protected action* [1] for the generation of a propagation identifier ensures the production of a unique identifier for each propagation instance within an application as part of the propagation *initialization* phase described in Section 2.

5 ANALYSIS

Our propagation model provides a design-time abstraction that focuses on the interrelationships between object types, as well as the ability to transfer interdependencies recognized at design-time to the implementation of the software system. Our use of Ada95's protected objects and tasking constructs provide support for the modeling of propagations so that the various phases of the propagation process can proceed concurrently. Our propagation-type construct provided a natural candidate for the introduction of concurrency into our model, via the Ada95 tasking constructs. Upon construction, a propagation entity requires non-serial communication with each of its source objects.

The use of objects with integrity constraints is critical to our propagation model. Our model effectively elevates the concept of integrity constraints from a matter internal to individual objects to a level at which consistency can be maintained at the multi-object level. Borrowing the concept of transaction rollback from database theory [8], our manner of handling IC-violating messages requires halting the triggering sequence, and restoring all of the interrelated objects to their respective pre-propagation states. Note that the former state of the source and destination objects are maintained by the PE for use in such a rollback situation. Other approaches exist, such as allowing the IC-consistent changes made to interrelated objects prior to the receipt of the IC-violating message to remain after the propagation terminates. Another approach might allow a propagation to continue execution after the receipt of an IC-violating message report from an interrelated object, effectively allowing all IC-consistent alterations to occur. We find both of these approaches incompatible with our goal of application-wide information consistency, since neither *guarantees* that the application remains in a consistent state when viewed from an application-wide perspective. With our approach to the handling of IC-violating messages reported during the propagation triggering phase, if the application was consistent (from an inter-object viewpoint) prior to the propagation, then after the propagation completes, the application remains consistent regardless of whether the propagation itself was successful or not. In the case of a successful propagation, all interdependent object-state alterations are accomplished under the guidance of the PE. In the case of a failed propagation, no interrelated objects states are altered as the net effect of the propagation.

6 RELATED WORK

Efforts in modeling the interdependencies between entities in the object-oriented development of a software system have focused on either the specification or implementation phases of software development. As part of the specification phase, the Unified Modeling Language (UML) [4] provide for the identification of object types involved in propagations, represented as "Interaction Diagrams". Like our propagation model, interaction diagrams support the identified during the analysis of a problem. Unlike like our approach, UML provides no provisions for carrying the identified object interaction through to the design or implementation phases of the software life-cycle. The Mediator [10] approach allows modularization of how software components work together into "mediators" which represent the behavioral relationships between independent components. Although similar to our work, the Mediator approach does not provide a precise mechanism through which a designer may specify interrelated message passing between objects, nor does it provide a mechanism to resolve system state inconsistency resulting from IC-violating message passing.

Progress has been made at the programming language level to provide programmers with the tools needed to better resolve consistency maintenance issues. The APPL/A [11] language adds the concept of programmable triggers to Ada 83. The programmable triggers of APPL/A provide a mechanism through which programmers may capture portions of inter-object relationships during implementation. Similarly, the R++ language [6] extends C++ with path-based rules that are triggered by changes to monitored objects. These path-based rules can be used to express multi-object methods that enforce invariance between multiple objects. Missing from the approaches taken by APPL/A and R++, as well as the related areas of database triggers, alerters and transactions [8], is a means through which the designer may specify which objects interact, when they interact, and precisely how the interaction takes place.

7 CONCLUSIONS

We have presented our research efforts on the expansion of the capabilities of the object-oriented paradigm to support application-wide information consistency, by providing a model through which we may specify the interdependencies between design objects. As part of this

effort, we have detailed ADAM, the object-oriented design environment within which our propagation model is embedded and presented a formalized object-oriented propagation design model. We examined our basis for a propagation-type, which included the phases and states that a propagation-type enters during its duration. We investigated the role of interrelated objects with respect to a propagation's execution, emphasizing the additional behavior requirements placed on objects by our propagation model. Such characterizations of propagation modeling are required as initial work towards a distributed model that communicates through CORBA [7]. We are currently developing a propagation-tailored IDL description to provide the foundation for CORBA-based propagation communication.

Another area of future work involves the extended-access of a propagation, with regard to its implicit involvement with other design objects. It is apparent that cycles may exist in chains of propagation-required message passing. It should be noted that the presence of a cycle does not necessarily indicate a poorly designed propagation. An interrelated object may need to query a second object during the course of propagation triggering, and the second object's response to this query may well be dependent upon the receipt of further, amplifying, information from the first object. While such behavior can currently be specified with our propagation model, further investigation is needed into a way to determine the conditions under which chains can be detected, and a means through which the ADAM environment can determine which of these chains require resolution by the propagation's designer.

ACKNOWLEDGEMENTS

D. Needham acknowledges, with appreciation, partial funding for this work received from the Naval Academy Research Council and the Office of Naval Research under N0001498WR20010. The views expressed herein are of the authors, not of the Office of Naval Research.

REFERENCES

[1] Department of Defense, *Ada 95 Reference Manual*, International Standard, ANSI/ISO/IEC-8652:1995, Jan. 1995.

[2] K. El Guemhioui, S. Demurjian, T. Peters, and H. Ellis, "Profiling in an Object-Oriented Design Environment that Supports Ada 9X and Ada 83 Code Generation", *Proc. of 1994 TriAda Conf.*, Baltimore, MD, Nov. 1994.

[3] H. Ellis and S. Demurjian, "Object Oriented Design and Analyses for Advanced Application Development - Progress Towards a New Frontier", *21st Annual ACM Computer Science Conf.*, Feb. 1993.

[4] M. Fowler, and K. Scott, *UML Distilled*, Addison-Wesley, 1997.

[5] M.-Y. Hu, S. Demurjian, and T.C. Ting, "Unifying Structural and Security Modeling and Analyses in the ADAM Object-Oriented Design Environment", in *Database Security, VIII: Status and Prospects*, J. Biskup, C. Landwehr, and M. Morgenstern (eds.), Elsevier Science, 1994.

[6] D. Litman, P. Patel-Schneider, and A. Mishra "Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules", *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications, OOPSLA'97*. ACM Press, Oct. 1997.

[7] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, John Wiley and Sons, 1995.

[8] P. O'Neil, *Database Principles, Programming, and Practice*, Morgan Kaufmann, 1994.

[9] T. Peters, S. Demurjian, D. Needham, R. Peters, S. Dorney, "Propagating Topological Tolerances for Rapid Prototyping", Proceedings of the International Mechanical Engineering Conference and Exposition (IMECE), MED-Vol 4, Atlanta, Georgia, Nov. 17-22, 1996, pp. 487-498.

[10] K. Sullivan. I. Kalet, and D. Notkin, "Evaluating The Mediator Method: Prism as a Case Study", IEEE Transactions on Software Engineering, Vol. 22, No. 8, August, 1996, pp. 563-579.

[11] S. Sutton, D. Heimbigner, and L. Osterweil, "Language Constructs for Managing Change in Process-Centered Environments", *Proc. of the Fourth Annual ACM SIGSOFT Symposium on Software Development Environments*, Irvine, CA, Dec. 1990.