An Analysis of Ada, Object-Oriented Design, and Structure Model as Implemented in a Moving Target Simulation Design

Thomas F. Flynn and Mary D. Petryszyn
CAE-Link Corporation
Binghamton, New York

ABSTRACT

Previous designs of moving target simulation models were developed using a single point of control functional architecture and functional design methodology. This approach to design concentrated on identifying all of the specific actions that would occur within a simulation software environment and relied on a single control point and a common data pool to provide sequence, control, and communications between all of these functions. Applying this to moving targets, the approach focused strictly on the specific actions that a unique platform must perform in order to satisfy an interface to other functions within the total environment. On the B-2 ATD program, it became apparent that designing math models representing functionality caused a number of problems. These problems involved: isolating control logic from system functionality; adaptability of software to accommodate future requirements, such as the addition of targets or modifications to the properties (geometry, weight and balance data, etc.) of targets; and taking advantage of software reusability. Also, the functional approach, since it dealt with specific actions, did not segregate basic platform structures and properties, platform functionality, and non-specific platform maneuvers from one another. These problems were the basic underlying reasons to use a more state of the art methodology and supporting language, Object-Oriented Design and Ada, to help reduce the functional approach weaknesses.

ABOUT THE AUTHORS

Thomas F. Flynn has been employed by CAE-Link Corporation for the last 18 years. During that time he has worked as a software systems engineer developing aircraft systems models for U.S. and foreign C-130 simulators, U.S. and foreign P3-C simulators, the F-4 simulator, the C-135B simulator, the initial phase 1 design and competition for the B-1B simulator, and the B-2 simulator. Over the last five years he has been involved in prototyping Object Oriented Design methodologies for the B-2 program and participated in the initial software architecture and management aspects of the software life cycle for the NASA Space Station simulator program.

Mary D. Petryszyn has been a Systems Engineer with CAE-Link Corporation for the last 6 years. She holds a B.S. in Electrical Engineering from Clarkson University and a M.S. in Computer Engineering from Syracuse University. Her present work includes systems engineering in the tactics simulation area for the B-2 Aircrew Training Device (ATD).

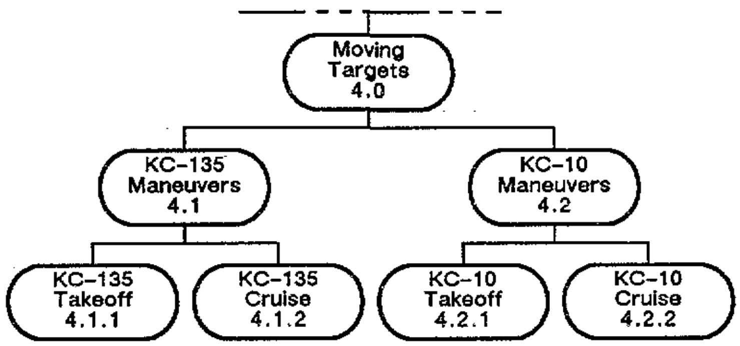
An Analysis of Ada, Object-Oriented Design, and Structure Model as Implemented in a Moving Target Simulation Design

Thomas F. Flynn and Mary D. Petryszyn
CAE-Link Corporation
Binghamton, New York

FUNCTIONAL DESIGN METHODOLOGY

This design approach involved the identification of functions representing major systems or sub-systems that were based on requirements from the customer device specification, derived analysis, or an experience base built up over years of system development. Each function identified represented a specific action for a specific device, system, or combination of systems to achieve some result that would be perceived by the student and/or instructor due to an external stimulus provided by another function, crew action, or instructor action. The initial product of this design phase was a software tree of actions that would produce the desired results of system requirements. The software tree only provided functionality (specific actions), not control or data structures relating to real-world components. The control was basically at one level, the system operating executive. The functions identified in the software tree were subroutines under control of the executive that would be invoked based on a master scheduling table. The executive provided a cyclic one-second period. The executive would update at the maximum rate the system required within that period. This method placed most, if not all, system control and knowledge at the single executive level within the master scheduling table. This required a very careful evaluation of when functions would be called within the table with respect to time, performance and other functions required to run within the same time domain. Some of the functions identified contained lower level control logic to account for mode control or special processing during simulation state changes. This control logic was embedded throughout the real time functionality of the system. For solution of closed loop problems found in areas such as aero, engine, and flight control functions, care had to be placed on where each of the subroutine calls occurred in the master table in relation to each other. Since all software entities in the tree represented actions, there was only one data structure defined within the total environment. This data structure was at the executive level and was represented as a common data pool to all functions in which they could read and write without any specific data consistency controls in place. This implied, in relation to an Object-Oriented design, that only one object existed, the total simulation environment.

In the case of moving target simulations, the top level requirement identified the need for actions of one or many target platforms. This requirement was based on the need to produce data for use by visual, radar, or environmental functions within the total simulation environment. This top level function was then decomposed into the specific actions for each of the specific moving target platforms defined in the specification. A typical software functional tree for the moving target simulation is shown in Figure 1. Each of the lowest level nodes shown in the tree represented a functional math model that would provide all of the required actions for the specific platform. As each math model was developed, specific and general platform functionality and the required properties were designed into the model. This meant, for example, that for KC-135 platform maneuvers involving taxi, takeoff, and cruise, each contained certain common functions or portions of functions, such as equations of motion, in order to satisfy that functionality. To go a step further, any additional vehicle platforms with a set of maneuvers contained the same or portions of the same common functionality. Specific platform properties such as weight and balance data were hard coded into each of the specific functions. The control of functions was local to the specific platform maneuver software. A platform maneuver or group of maneuvers would be invoked based on triggers or common data pool information.



ITSC_ITEC

Figure 1 Functional Architecture

OBJECT-ORIENTED/BASED METHODOLOGY

This methodology takes a different initial view of the design problem as compared to the functional approach. The major advantages to the Object-Oriented approach are the reduction of complexity, reuse of software, reduced testing of software, and a more localized effect of modification to software. The object-oriented methodology attempts to define smaller data structures that define attributes representing the state of an entity in a static situation. This process attempts to reduce the complexity of the data structures down from the single structure, which is very complex and used in a functional approach, to structures specific to the entities in the problem space. This means that the problem space is evaluated for components that possess state information about themselves, and to isolate each occurrence of a component based on minimum dependencies between one another. An example would be an airframe, the environment, targets, etc. In each of these examples, there is information that is required between each of these components but there is not common functionality between them. Each of these entities represent state and will contain operations that will modify that state based on external stimulus.

In the Object-Oriented view, the actions identified in a specification or derived through analysis are used to define the level of fidelity of the entity's behavior. In defining these components it becomes apparent that similar occurrences of the component are found. These similarities are identified by groups that can be reduced down to a base class or type for the entity. Each occurrence will contain the same static attributes representing state and will exhibit the same behavior required to change it's state due to an external stimulus. What makes each occurrence different is its name and its unique set of properties. Given one set of operations, one object's properties in the form of constants or lookup tables will cause that set of equations to act differently from that of another object with a different set of properties but the same equations.

Ideally, each common set of objects and the resulting base class will relate one-to-one to objects in the problem domain such as a battery, a valve, or a pump. Some objects and the resulting class represent a very complex structure and need to be reduced simply because of the complexity or because of typical modifications that may occur over the life of the software. An example is the airframe. In reducing the complexity, the objects and classes are more difficult to identify since they may not relate to a physical component in the problem space. This situation requires evaluation of all of the component's state data in relation to dependencies on one another. Some of the more critical dependencies evaluated are time domain problems in which certain groups of state data must be evaluated within the same functionality. These dependencies are grouped within common data structures that will relate to a single object or base class with the required operations to change that state data. The attempt is to isolate these dependencies into groups of data structures and associated behavior which translate into the less complex classes.

Another consideration in this evaluation, but second to the time domain problem, is the format in which the data used to identify the properties of an object or class is typically received from an airframe manufacturer. In this case, the reduction of the data structures (objects or classes) is determined on the logical grouping of the data used to identify the properties. This typically will only work where it does not create strong dependencies between objects or classes.

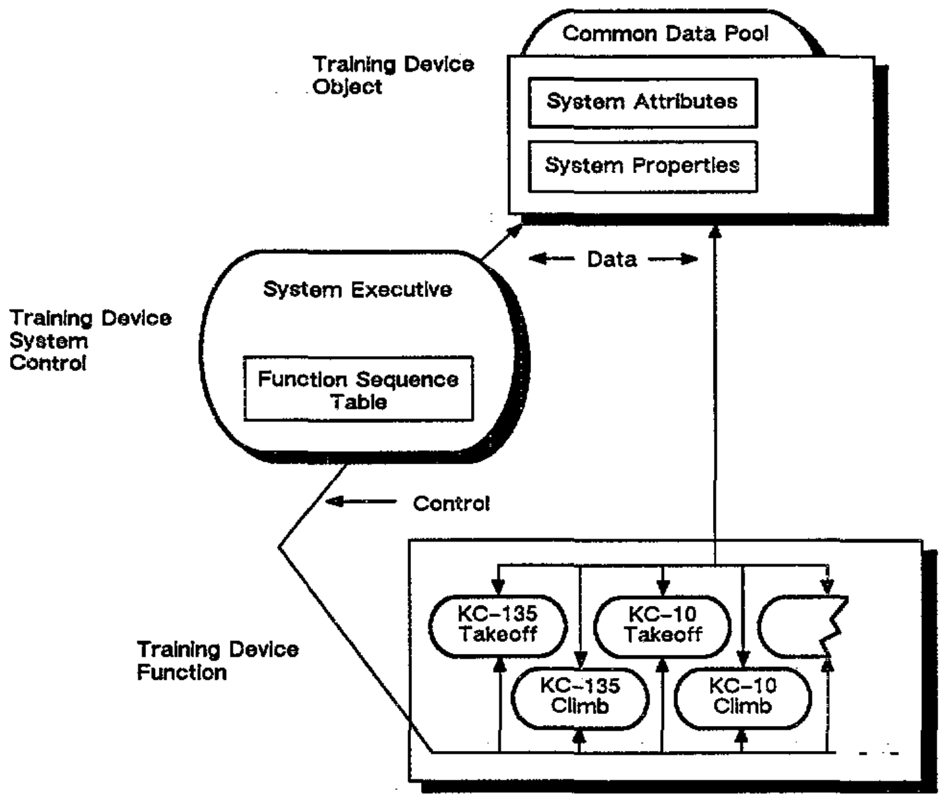
Based on these points, the Object-Oriented approach clearly shows how complexity is reduced by localizing data structures and reducing interdependencies among those structures. By defining a base class for use in creating instances of objects, the reusability aspect becomes a strong point in that only the base class is developed by the software team and, the resulting class or type is used to create each instance of the object within the environment. This feature also reduces lower level testing and the amount of regression testing. For classes that contain operations with high levels of complexity, the testing can be performed on the base class with a range of properties typical of the instances to be used in the simulation environment. Finally, modification to software will either involve changes to the fidelity due to new requirements, or design problems, or changes to the properties of an instance to a class. Modifications to fidelity can be changed and tested at the base class level. The result with changes to operations is an automatic change to every instance of that base class used in the environment. In some cases, the testing of the change can be kept at the base class level and will satisfy all instances of that class. Where the properties of an instance change, only that instances property values need changing. This eliminates the need to modify anything in the base class, data structures or operations, and as a result, localizes the change to the specific instance. The one drawback of the class or type approach to representing the template of a group of objects is that the data structures and the operations must represent the worst case fidelity requirements in order to satisfy all instantiations of the class.

The resulting objects developed through this analysis represent a static set of structures that require some controlling mechanism to provide an orderly set of input stimulus based on events occurring in the simulation environment. The objects themselves should not have any knowledge of events outside of their own domain in relation to their attributes, so as to keep complexity down and to prevent carrying special processing and/or interdependencies from a base class to all instances of the class. This requires a control mechanism to evaluate the state of the environment based on events provided by a high level service routine to determine when to invoke an object's operation and what parameters to pass it.

The control mechanism also contains areas to keep data imported from external systems and data to be exported to external systems. The object's behavior is intended to represent a real time solution in relation to it's state data. This behavior requires a set of input parameters from other objects, sub-systems, or procedures at the system level during a real time solution. For non-real time, the system may be required to stabilize each of it's objects for initialization to a pre-defined point or establish special mode control and special processing for auto test case control. This stabilization can be designed into the base classes as a separate procedure or can be performed by the system controller directly on the objects' attributes. The structure models used in the object oriented approach show groups of systems that contain a collection of objects associated with the system. The system controller is intended to contain all knowledge of how that system is to be controlled through procedures and invocation of it's objects' operations. The controller may contain procedures that represent actions or decisions that must be passed on to it's objects due to external events in the environment.

As was shown in the functional structure, actions were embedded into each specific function such as *KC-135 climb and KC-135 cruise. The controller in* the object oriented structure may have general procedures representing climb and cruise that work on a standard set of inputs required by a KC-135 object. These procedures may only provide a control command to climb or cruise to the object. The same procedure and inputs would be used for a KC-10. Since the objects contain very different properties even though they both have identical functionality, they will respond to the commands differently due to the property difference.

The top level executive in the Object-Oriented structure has no knowledge of how any system is to be controlled. The executive only knows the sequence and rate at which the system controller will be called and provides the system with the environmental parameters or events to allow it to respond relative to the state of the environment. This is very different from the functional structure in that the functional executive contains all knowledge of systems and must control the sequence and dependencies of all functions within the environment as is shown in Figure 2. It becomes apparent that the complexity of a functional executive can increase exponentially as more functionality is added throughout the development of the system. The Object-Oriented structure helps to



ITSC_ITEC

Figure 2 Functional Structure Moving Target Simulation

localize complexity to the system level while keeping the executive at a fairly constant level of complexity.

The independent system development of the Object-Oriented structure also allows for complete independent testing of the system in a non-integrated mode for all modes or states that may be encountered in the environment. This allows a major system to be integrated into the software environment incrementally and with a high degree of confidence that it will respond as designed. Figure 3 represents an object oriented system for the moving target simulation.

MOVING TARGET OBJECT-ORIENTED DESIGN

The real-time functionality of any system must be *addressed at some level. By using the main objectives* of reducing complexity and developing the most efficient design possible, the level at which the functionality of a system appears varies. In this design, requirements included multiple moving targets performing a variety of automatic motion maneuvers. A data structure was developed to define attributes and properties of the moving target object class. The attributes represent the object's state, including where it is going and what it is doing in the environment. The properties represent the moving target's motion performance characteristics.

General procedures were defined that were independent of the target class to represent all unique portions of the desired maneuvers, such as taxi, takeoff, and cruise, so as to remove any common or repeated functionality that is typical of a functional architecture. This works well because the input and output parameter names defined for each maneuver remain constant, only the computed value of the standard commanded output parameter changes. Based on the *state of the environment* as defined by external imports or executive data and the object's specific attributes, the appropriate maneuver procedures are invoked by the system controller. This keeps the procedures isolated from any knowledge of the system or events in the environment, constraining their visibility to the specific input and output parameters upon which they operate.

Figure 4 shows the above mentioned automatic maneuver procedures with their associated parameter lists. Note that the output parameter names of ordered position (i.e., the position the moving target must drive to) are the same for each procedure. This allows a standard input parameter list for the target equations of motion operations in relation to maneuver commands. This also allows additional maneuver operations to be developed without affecting the existing parameter and data structures. Since each specific maneuver function contains the same named out parameter, which is passed by the controller to the target, only the computed data contained in the parameter will be different based on the specific maneuver invoked by the controller. Only one of these procedures is invoked by the system controller for each specific moving target at any given time and the output parameters are then used to drive the equations of motion.

General procedures for manual maneuvers were designed to implement instructor motion and position commands, such as commanded range and relative *bearing changes or updated altitude and velocity* commands on an object in the system. The manual maneuvers allow the instructor to deviate from an automatic profile based on dynamic conditions occurring within the training environment. These procedures were developed to represent each unique type of manual maneuver required to satisfy the range of dynamic instructor controls and are invoked by the system controller based on instructor events passed into the system through the system importer. As in the automatic maneuver design, the input and output parameter names remain the same as those in the automatic procedure for each of the manual maneuvers and limit the knowledge required by the procedure to only the attributes of the specific object necessary to its operation.

Figure 5 shows a sample of the manual maneuver procedures with their associated parameter lists. Note that the output parameters of ordered position are the same as those shown for the automatic procedures in Figure 4. In these cases, the mode of the moving target is manual as determined by events in the environment, which overrides the automatic mode of a moving target until manual mode is no longer desired. These manual maneuvers are invoked by the system controller based on a manual maneuver event from the instructor, and a subset of the parameters are modified that correlate to the manual commands received. Other procedures shown in Figure 5 cause an instantaneous change to the state of the moving target, and are therefore invoked by the system controller in addition to the associated automatic maneuver for each specific moving target. These procedures are invoked once for each separate event in the environment, the mode remains automatic and the output parameters are the current position of the moving target. In both cases the output parameters of

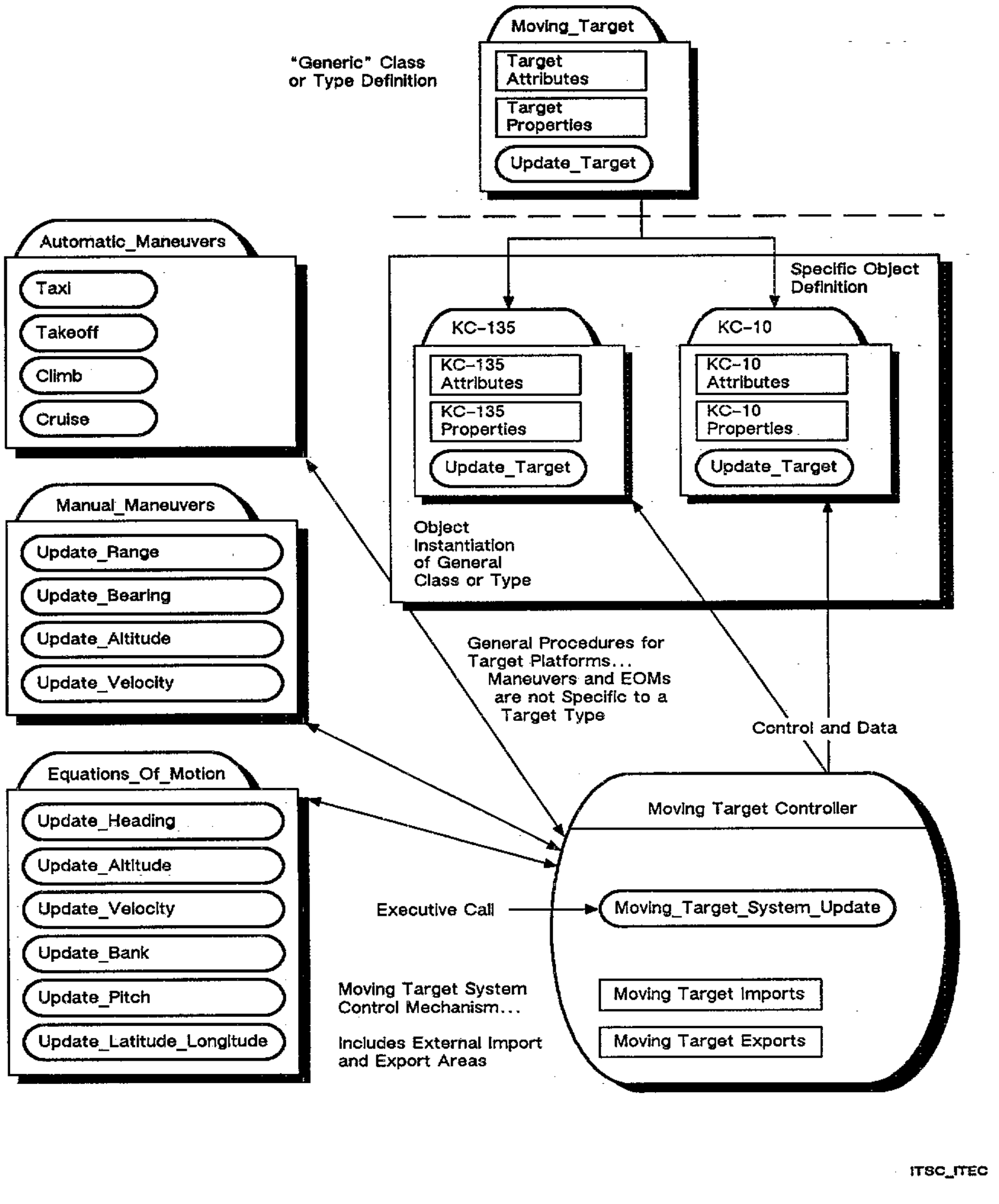


Figure 3 Object-Oriented Structure Moving Target Simulation

```
Procedure taxi      (

    taxi_profile        : in taxi_automatic_profile;
    current_latitude    : in latitude_type;
    current_longitude   : in longitude_type;
    current_altitude    : in altitude_type;
    current_heading     : in heading_type;

    ordered_latitude    : in out latitude_type;
    ordered_longitude   : in out longitude_type;
    ordered_altitude    : in out altitude_type;
    ordered_heading     : in out heading_type )  is
                            .
                            .
                            .

end taxi;
```

```
Procedure takeoff   (

    takeoff_profile     : in takeoff_automatic_profile;
    current_latitude    : in latitude_type;
    current_longitude   : in longitude_type;
    current_altitude    : in altitude_type;
    current_heading     : in heading_type;

    ordered_latitude    : in out latitude_type;
    ordered_longitude   : in out longitude_type;
    ordered_altitude    : in out altitude_type;
    ordered_heading     : in out heading_type )  is
                            .
                            .
                            .

end takeoff;
```

```
Procedure cruise    (

    cruise_profile      : in cruise_automatic_profile;
    current_latitude    : in latitude_type;
    current_longitude   : in longitude_type;
    current_altitude    : in altitude_type;
    current_heading     : in heading_type;

    ordered_latitude    : in out latitude_type;
    ordered_longitude   : in out longitude_type;
    ordered_altitude    : in out altitude_type;
    ordered_heading     : in out heading_type )  is
                            .
                            .
                            .

end cruise;
```

Same output parameters for each procedure --

Different values based on maneuver executed

Ordered position output parameters used to drive equations of motion procedures

Figure 4 Automatic Maneuver Procedures

```
Procedure update_altitude          (
      manual_altitude    :  in altitude_type;
      current_altitude   :  in altitude_type;

      ordered_altitude   :  in out altitude_type  )  is
                                 .
                                 .
                                 .
end update_altitude;
```

```
Procedure update_velocity          (
      manual_velocity    :  in velocity_type;
      current_velocity   :  in velocity_type;

      ordered_velocity   :  in out velocity_type  )  is
                                 .
                                 .
                                 .
end update_velocity;
```

-- Manual mode overrides
-- Output parameters same as
   automatic maneuver procedures

```
Procedure update_range             (
      manual_range       :  in range_type;
      current_bearing    :  in range_type;
      ownship_latitude   :  in latitude_type;
      ownship_longitude  :  in longitude_type;

      current_latitude   :  out latitude_type;
      current_longitude  :  out longitude_type  )  is
                                 .
                                 .
                                 .
end update_range;
```

```
Procedure update_bearing           (
      manual_bearing     :  in bearing_type;
      current_range      :  in range_type;
      ownship_latitude   :  in latitude_type;
      ownship_longitude  :  in longitude_type;

      current_latitude   :  out latitude_type;
      current_longitude  :  out longitude_type  )  is
                                 .
                                 .
                                 .
end update_bearing;
```

-- Automatic mode maintained
-- Instantaneous position changes
-- Output parameters same as
   equations of motion procedures

Figure 5 Manual Maneuver Procedures

these system level functions are used as input parameters to drive the equations of motion.

Finally, the equations of motion math models were designed such that they are independent of specific platform functionality and properties, and will respond to any of the automatic and/or manual system level functional commands since the command input parameters are the same. As the math model for the class was developed, the necessary specific object properties (geometry, weight and balance data, etc.) were defined as input parameters as opposed to being hard coded, thereby making the equations of motion adaptable to the specific properties of each object and also eliminating repeated common functions or portions of functions. By also including the object attribute that defines the necessary integration rate for an object based on the state of the environment, the equations of motion math models can respond without having direct visibility to these events.

Example Procedure X was developed and used in this moving target design to compute velocity and acceleration, which are functions of time. It is shown in pseudo-code for ease of discussion and understanding. By utilizing the parameter passing feature of the Ada programming language, the procedure was developed so as to define all moving target specific attributes and properties needed by the math model as input and/or output parameters. The same named parameter lists are passed in and out for each specific instance of a moving target class; however, the data contained in the specific parameters is different between one instance and another. This allows the behavior of an instance of the moving target class to respond uniquely based solely on the input values of the specific properties. Thus, it is a general procedure that can be used to compute the velocity of any of the moving targets in this design. The integration rate is passed as a parameter, making it possible for the controller to determine the rate necessary for each particular moving target based on events in the environment.

In contrast, Example Procedure Y represents a typical comparison of the same procedure for previous, functional moving target designs. This procedure is specific to the moving target being simulated, a KC-10 in this case, with the properties of this moving target being hard-coded into the math model. Also, since the rate at which this procedure is executed is constant as controlled by the top-level exec, the integration rate is constant and defined in the procedure. Since a different and separate procedure similar to that of Example Procedure Y would be used for each specific moving target, such as a KC-135, it is easy to see the impact of making any changes to a math model in this type of structure. For instance, a change to the integration rate would cause the same change to be made to both procedures.

As shown in the object oriented approach, the command and control for the moving target, which are the maneuver functions, has been isolated from the functionality of the target class. The functional approach shown in example procedure Y does not contain this separation of command and control from the target functionality and, as a result, forces the maneuver functions to be embedded into each specific moving target function.

As can be seen in Figure 3, the system controller determines the actions and provides a corresponding set of inputs to the object based on the state of the environment. The object's attributes and specific events indicate to the controller which maneuver procedures and equations of motion need to be invoked, and how often. With this structure, the individual simulation fidelity of each moving target can be independently and dynamically controlled. For instance, it is not necessary to update a moving target object's position as often or with a high fidelity when it is not visually seen. This information has no affect on the actual operation of the maneuver or equations of motion procedures and is only used by the controller to make the necessary and proper execution decisions for an object.

Now, looking at potential modification of the software, it is easy to see that a change to a maneuver or an equation of motion would involve only making the change in one place due to the removal of common functionality. The affect would automatically be imposed on every moving target object in the system. Testing of the change could be performed on an object with properties typical to the simulation environment, since the change is not specific to an instance of a moving target object but is a change to the class of moving target objects. Modification of the specific properties of one or more moving target objects would not require any software modification to the maneuvers or equations, and would have no affect on the other moving target objects in the system. Computer loading can be more easily determined with this type of structure for target or maneuver additions, since worst case timings are available for typical targets and maneuvers currently simulated in this design. This gives a more accurate estimate of change impact that was not available with previous types of design structures.

```
Procedure compute_velocity_and_acceleration (

    delta_time                     : in seconds;
    K_values                       : in times;
    performance_characteristics    : in characteristics;
    ordered_velocity               : in feet_per_second;
    past_velocity                  : in feet_per_second;
    past_acceleration              : in feet_per_second_squared;
    current_velocity               : in feet_per_second;
    current_acceleration           : in feet_per_second_squared  )  is

begin

    -- compute the difference between ordered and current velocity

    $\Delta v = v_{ord} - v_c$

    -- compute the new current velocity

    $v_c = v_{ord} - \Delta v K_1 + ( v_c - \Delta v K_2 ) K_3$

    -- compute the new current acceleration

    $\dot{v}_c = \Delta v K_4 + ( \dot{v}_c - \Delta v K_2 ) K_5$

    -- limit the computed acceleration within performance characteristics

    $\dot{v}_{min} < \dot{v}_c < \dot{v}_{max}$

    -- recompute the new current velocity after limiting the acceleration

    $v_c = (v_c)_{n-1} + \Delta t \dot{v}_c$

    -- limit the computed velocity within performance characteristics

    $v_{min} < v_c < v_{max}$

end compute_velocity_and_acceleration;
```

Example Procedure X

```
ordvel                                    Common Data Pool
pastvel
pastaccl
currvel
curraccl
```

```
kc10vel:
    data:   dtime = 0.03125
            Kvalues = $(K_1, K_2, K_3, K_4, K_5)$

    -- compute the difference between ordered and current velocity
    $\Delta v_{ord} - v_c$
            -- ordered and current values in common data pool

    -- compute the new current velocity
    $v_c = v_{ord} - \Delta v K_1 + ( v_c - \Delta v K_2 ) K_3$
            -- K values constant since dtime

    -- compute the new current acceleration
    $\dot{v}c = \Delta v K_4 + ( \dot{v}_c - \Delta v K_2 ) K_5$

    -- limit the computed acceleration within performance characteristics
    $1000.0 < \dot{v}_c < 9000.0$ -- constant performance characteristics

    -- recompute the new current velocity after limiting the acceleration
    $v_c = (v_c)_{n-1} + dtime\ \dot{v}_c$
            -- past and current values in common data pool

    -- limit the computed velocity within performance characteristics
    $100.0 < v_c < 900.0$              -- constant performance characteristics

return
```

Example Procedure Y

Take, for example, the case of adding the requirement for a new type of moving target to perform automatic maneuvers representing evasive actions to the system shown in Figure 3. Also, assume an additional requirement for response of an object to instructor manual maneuver commands of updated latitude and longitude. This modified system structure is shown in Figure 6. It is easy to see here that the changes made to the system had no impact to the maneuver simulation already in place. The data imported would now also include the additional instructor manual commands. The addition of a new type of moving target object would only involve defining its specific properties, similar to changing the properties of an existing type of moving target object. The system controller would be the centralized point in the system to experience the main effect of the updates , since it has the visibility to the new maneuvers in the system as well as a connection to the new events in the environment.

Testing of the changes made would involve having to test only the new procedures, since the specific attributes of an object are defined as parameters. As in the example of changing an existing maneuver, testing of the new maneuvers can be performed on an object with typical simulation properties and attributes. Not only is this system highly maintainable due to these factors, it is also easily reusable.

Since this system structure contains its own total system level functionality and system level control mechanism, the top level executive can be isolated from the knowledge of how the system must be controlled. It is easy to see that this system could be reused by simply making the proper connection of the system controller's external import and export data structures, and telling the top level executive the sequence and rate at which to invoke the system controller, and the environmental parameters and events to pass it.

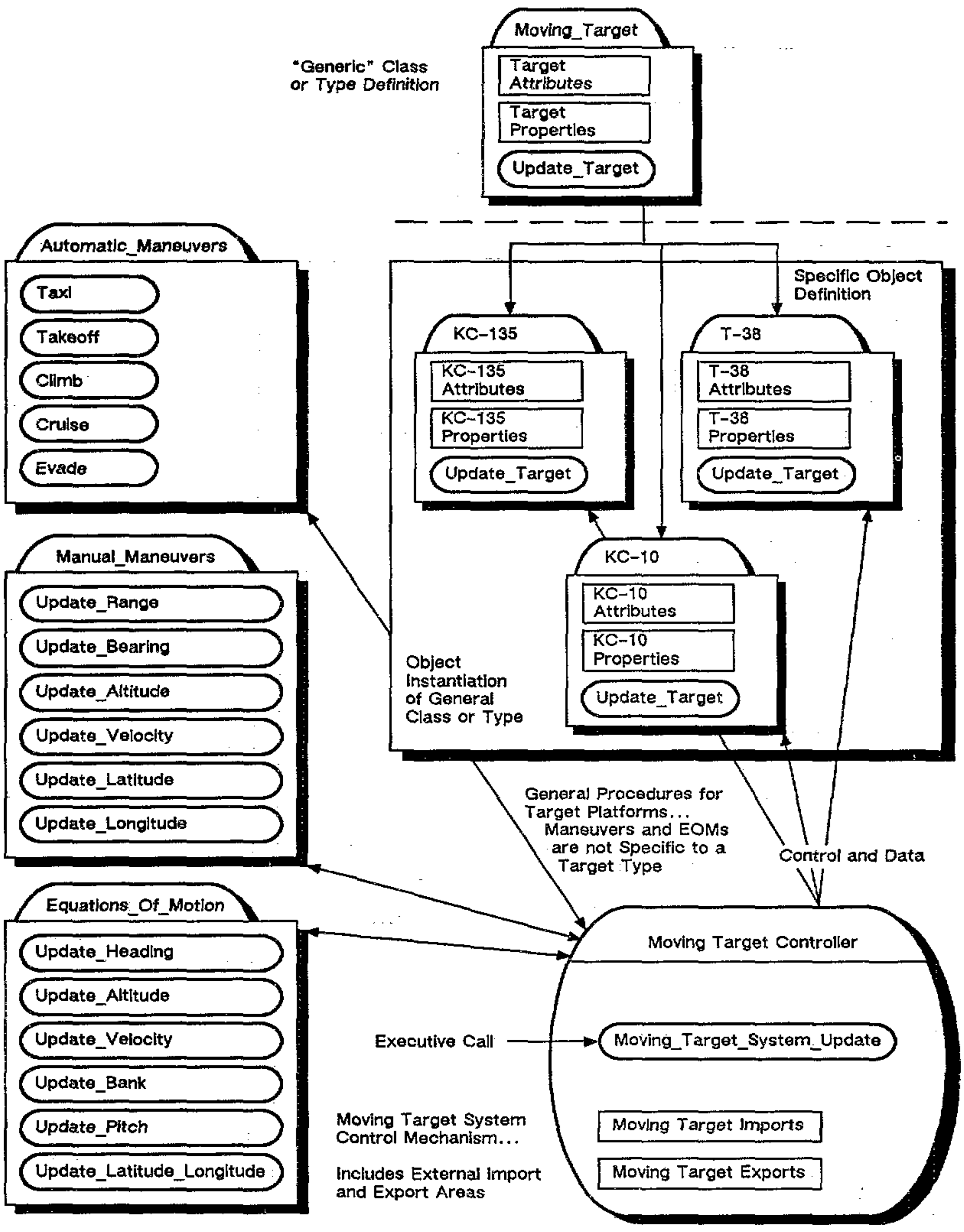
CONCLUSIONS OR FUTURE OUTLOOK

The Object Oriented architecture has solved a large majority of design problems encountered with the functional approach for the more complex training devices in today's market. These problems involved isolating the events and control of a system for both real time and non-real time modes from the functionality of objects within the system and has reduced highly complex structures into smaller, well defined, and less complex tasks. It has also taken advantage of re-use of software products within the existing environment and allows for that re-use in future products. This approach reduced lower level testing of functionality by allowing testing of the base class of an object instead of each instance of that object. By grouping dependencies within common data structures that relate to a single object or base class, higher level testing is reduced to testing of interfaces between objects.

One new direction to consider for improved efficiency is removing the polling of events at the system controller level, and pulling it up to a service routine approach at a higher level through the simulation exec. This should be done so as to develop a standard set of event driven entry points into systems. This way, systems can be developed that are adaptable to different simulation environments. Potential candidates for efficient use of this concept include initial condition (IC) and reset implementations, malfunctions, and Instructor Operator Station (IOS) features that are true events such as mode controls, weather changes, and moving target commands.

The IC implementation used on the B-2 ATD program allows use of this concept in that systems can contain controllers to perform special processing for an IC, that are only called by the simulation exec for this type of event. With the IC implementation isolated to the system controller, the controller can make a decision to use a real-time solution to drive the system to a stable state or to directly stabilize the system to a pre-defined state based on the parameters received.

The structure model presented in this paper has shown it's capability to adapt to changes and advances in the area of flight simulation. The architecture represents a dynamic concept that can be adapted to advances in the state of the art. It embodies a basic plan to take advantage of advances in command and control techniques, real time system functionality, interfacing techniques, and computational hardware developments.



ITSC_ITEC

Figure 6 Modified Object-Oriented Structure Moving Target Simulation