DDD v4.0 Scenario Writing Guide

December 13, 2006

Contents

Chapter 1: Introduction

This document describes an early release of the Advanced Scripting for Training, Research and Analysis Language (Astral). This language is the means by which an experimenter describes the scenario on which a simulation "game" will be based. Thus, the primary audience consists of professionals involved with research or training using Aptima's Distributed Dynamic Decision-making system (DDD). They are expected to understand the function and uses of the DDD. (For an overview of the DDD see the *Best Practices Guide*). The secondary audience consists of students and IT professionals who may be drafted to assist with the writing of a scenario.

Scenarios are written in XML format; a familiarity with the concepts of XML will be very helpful in reading this document. See http://www.mousewhisperer.co.uk/xml_page.html.

What is the DDD?

DDD is a flexible and powerful computer-based simulation tool for creating and conducting research and training activities involving cooperating teams of decision-makers.

A simulation using the DDD involves a number of *assets*, some controlled by *players* and some autonomous, that interact on a defined *playfield*. The players must accomplish a specific task using these resources, and the communication pathways between players are typically constrained by the simulation. The actions of the players are recorded for later analysis, review, or playback.

The simulations defined by DDD scenarios are often called *games*, because they have many of the features of multiplayer computer games. That practice will be followed here.

The Skinny on Scenarios

A scenario is a description of all the elements of a game. In particular, a scenario defines, at a minimum, the following:

- The playfield on which the game is played
- The roles chosen by the players
- The structure of the organizations to which those role players belong
- All assets that appear (or might appear) in the game

In most scenarios, there are assets that are not controlled by any player. The scenario also describes the behavior of those assets during the game.

Scenarios need not be highly complex, although even the simplest require a significant amount of specification of the elements of the game. On the other hand, Astral provides many options that allow for highly specific definitions of the assets and their interactions. This chapter and the next present the simplest scenarios.

XML – The Scenario Writer's Swiss Army Knife

Astral is written in a dialect of XML. XML is best understood as a framework for structured text, where the structure is imposed in part by "tags" in the XML and in part by the reader of the document.

For example, if an address book were organized using XML, a scenario writer might end up writing an address like the following:

```
<Name>Pino's Pizza</Name>
<Address>
    <Number>23</Number>
    <Street>Beacon St.</Street>
    <Locale>
        <City>Brookline</City>
        <State>MA</State>
        <Zip>
            <Five>02167</Five>
            <Plus 4>2819</Plus 4>
  </Locale>
</Address>
```

The items within pointy brackets are called Tags; every opening tag (like <Address> must be matched by a corresponding closing tag (</Address>). Also note that if an opening tag appears between a pair of opening and closing tags, then so does its closing tag; thus, since the opening tag <Locale> appears between <Address> and </Address>, the matching closing tag </Locale> must also appear between them.

The XML shows us that an entry is created from a name and an address, and that the address is described in terms of its individual fields—and that some of those fields have their own sub-structure. The XML does not show how these entries will be stored, used, or even interpreted; while names like "Locale" and "State" are suggestive, this same structure could theoretically–and quite pointlessly–be used for many kinds of structured data. So, users might see a lamp as having a shade ("<Name>" and a body ("<Address>", a

body having three parts–a wire (“<Number>”), a bulb holder (“<Street>”), and an on-off mechanism (“<Zip>”), and so on. There is no good reason to ever misuse a set of tags in this way, because the XML language does not care what names are used in tags–users can make them up for whatever application they have. All of the meaning is supplied by the interpreter of the XML description. In the case of Astral, the interpreter is the DDD Server.

The Life-cycle of a Scenario

The very first step in creating a scenario is planning. The details of the scenario should be carefully specified before any attempt is made to commit it to electronic form. This advice is usually given for documents that will have an electronic representation. It is given here for the following specific reasons:

1. The use of XML, together with other technical considerations, has given Astral an inflexible syntax; even small variances from standard form will cause the scenario to be rejected. Among Astral’s inflexibilities is case sensitivity. Thus, <Name> and <name> are not the same tag
2. In advanced scenarios, the richness of the options that Astral presents for specifying both assets and actions often requires that the characteristics of different asset types be carefully coordinated.

Once the scenario has been planned, it should be committed to electronic form. The format of a scenario is specified by an XML schema, which is provided for scenario writers. The XML description of a scenario can be written in many different tools, so long as the form in which it is passed to the server is simple text. A text editor (such as Microsoft’s Notepad) may be a better tool than a word processor (such as Microsoft’s Word) because the latter includes formatting in its documents, and that formatting would make the document unreadable as XML. Word processors *can* save documents as simple text, but using an editor designed specifically for XML is better..

A partial or completed scenario becomes input to the DDD Simulation Server console. The console will reject an ill-formed scenario. The console can be used to check a scenario without running a simulation. Once the experimenter is satisfied with the scenario, it is used as the basis for a run of the game.

What Next?

Chapter 2 will describe the essential features of any scenario, and will perform a close analysis of a simple scenario. Subsequent chapters will discuss these features in more detail.

Chapter 2: Overview of a Simple Scenario

Telling a story

Most scenarios tell a story. More specifically, they describe the physical objects that make up the background of a story, and may specify the behavior of those objects. The remainder of the story–both the context and the goals of the players–is typically given outside of the scenario as part of the experiment, and the working out of the story is defined by the actions of the players.

Consider an example of scenario writing that begins with a very simple story. There are two jet planes belonging to two different parties: Red and Blue. The Red plane is a Mig-15 and the Blue plane is the equivalent Shenyang J-4. Each has some kind of fixed weapon (for example, a gun or a laser) that is 70% effective at distances up to two kilometers. They are about to battle over the Korean Peninsula. Each has a radar system with a range of 3.2 kilometers. The Red Plane is flying back and forth provocatively between Pyongyang and Seoul, while the Blue Plane begins by taking off from Busan to intercept it.

Note that it is necessary to supply to the DDD a map of the locale of the game. Because the map might be just a section of a larger map, it is important to also give the point on the map that is coincident with the lower left corner of the screen. The location is described in UTM coordinates. Thus, we will provide a UTM grid identifier – S-52 in this case, and the offsets of our origin from the lower left portion of the grid. As Pyongyang is very close to the western border of the grid, the Easting is assumed to be 0, which means that the playfield starts at the Western edge of the grid.

The vertical and horizontal scaling values are in meters/pixel

. Figure 1 shows the actual playfield.

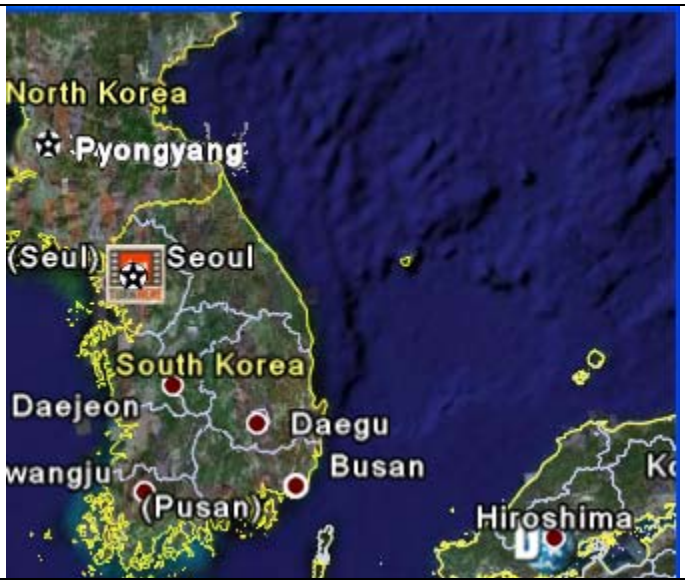


Figure 1

Also, it is necessary to specify some characteristics of the fighter jet. For this scenario, it is necessary to note the maximum speed of 1075 kilometers per hour. The weapons capabilities of the plane were discussed earlier in this guide.

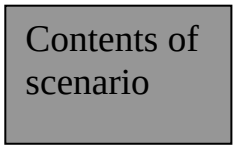
Finally, it is necessary to note the locations of the features that are referenced in the scenario. All distances are taken from the lower left corner.

City	Easting	Northing
Pyongyang	92	658.1
Seoul	206.8	383.1
Pusan	246.8	99.4

The structure of scenarios

With these details settled, it is possible to write the bulk of the scenario. Scenarios are described in XML, which means creating a tree-structured hierarchy of nested balanced tags. The outermost tags simply specify that there is a scenario:

<Scenario>

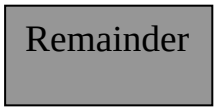


</Scenario>

The scenario must have a few key components. The scenario has a name and description that may be displayed on the game screens. Also, it is necessary to provide information about the field of play.

<Scenario>
<ScenarioName>…</ScenarioName>
<Description>…</Description>
<Playfield>

</Playfield>



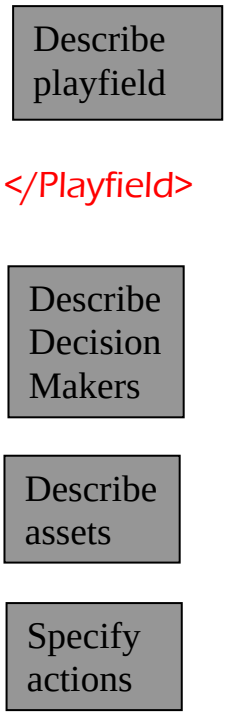
</Scenario>

Many of the items that come after the Playfield are optional. However, the definitions of the players are required. Players are officially called “Decision Makers.” Decision Makers represent virtual or potential players that might or might not be represented by live people during any particular run of the game.

The bulk of what remains in the scenario is the description of the assets. This is a very complex area that will be explored in a number of small steps. Finally, there are instructions that specify what the assets do. These instructions are primarily for assets that are not controlled by human players.

Players can direct the assets to move around, following specific paths on the playfield. Thus, the structural diagram of a scenario–which ignores certain details, to be discussed later–looks like the following:

<Scenario>
<ScenarioName>…</ScenarioName>
<Description>…</Description>
<Playfield>



The remainder of this chapter describes how to fill out those blocks in the context of the sample scenario.

Scaffolding

As noted, the greater part of scenario writing involves specifying the assets used in the scenario. Those details will be described later in this guide. This discussion begins with simpler but very necessary elements.

The first element is the outermost shell, as shown below:

```
<Scenario>
    <ScenarioName>Sample</ScenarioName>
    <Description>Intercept red plane flying over Korea</Description>
</Scenario>
```

There are some guidelines that must be followed. First, the case of words within the brackets matters. For example, <Scenario>…</scenario> would be rejected because one word is in upper case and the second word is in lower case. Second, when there is potentially long text to write, such as the description, do not insert any end-of-line characters. In particular, do not

press Enter inside the text. As shown in the previous example, the scenario proper begins with a specification of the playfield, as shown below:

```
<Playfield>
  <Map Filename>KoreaMap S 52. jpg</Map Filename>
  <UTMZone>52-S</ UTMZone >
  <VerticalScaling>318</ VerticalScaling >
  <HorizontalScaling>0</ HorizontalScaling >
</Playfield>
```

Note that the mapfile is a jpeg that must reside on the *client* machine, meaning, the PC used by the player. At present, the map must be approximately 4300 pixels wide and 3670 pixels high.

This specification goes after the </Description> and before the </Scenario>, so the complete scenario thus far looks is as follows:

```
<Scenario>
  <ScenarioName>Sample</ScenarioName>
  <Description>Intercept red plane flying over Korea</Description>
  <Playfield>
    <Map Filename>KoreaMap S 52. jpg</Map Filename>
    <UTMZone>52-S</ UTMZone >
    <VerticalScaling>318</ VerticalScaling >
    <HorizontalScaling>0</ HorizontalScaling >
  </Playfield>
</Scenario>
```

A *region* is a patch of land, sea, or some three-dimensional topographic feature. There must be at most one *region* specified, otherwise there is no field on which to play. The simplest type of region is a *Land Region*. A land region gives the outline of a Land Area. An asset that can only exist on land cannot move out of Land Areas. Similarly, a submarine or whale cannot successfully make its way onto land.

A region is specified by giving points on its outline–the points are considered to be connected by straight-line segments. An outline is best determined by using a digitizing program. Such a program makes it possible to click on points and will show the coordinates (usually in any scale preferred). The outline need not be overly precise. This is a decision that the experimenter makes, but in most uses of the DDD, it has been acceptable for the boundary to be somewhat approximate.

Shown below, for example, is the boundary to be used for the Korean peninsula:

Every region must have a name. This sample scenario will use the following name for this region:

```
<ID>Korean Peninsula</ID>
```

The region is specified in the scenario by giving the list of points. Each point is specified as an XML unit in the following way:

```
<Vertex>
   <X>6418</X>
   <Y>16012100</Y>
</Vertex>
```

The following is the specification for the peninsula. Note that the points run clockwise from the top left in this case.

```
<LandRegion>
  <ID>Korean Peninsula</ID>
  <Vertex>
    <X>1305.050</X>
    <Y>492731.000</Y>
  </Vertex>
  <Vertex>
    <X>131555.000</X>
    <Y>495060.000</Y>
```

```
</Vertex>
<Vertex>
  <X>79998.600</X>
  <Y>444510.000</Y>
</Vertex>
<Vertex>
  <X>86778.900</X>
  <Y>421986.000</Y>
</Vertex>
<Vertex>
  <X>83456.700</X>
  <Y>397680.000</Y>
</Vertex>
<Vertex>
  <X>167983.000</X>
  <Y>234975.000</Y>
</Vertex>
<Vertex>
  <X>179371.000</X>
  <Y>151747.000</Y>
</Vertex>
<Vertex>
  <X>167152.000</X>
  <Y>82291.600</Y>
</Vertex>
<Vertex>
  <X>131300.000</X>
  <Y>40488.100</Y>
</Vertex>
<Vertex>
  <X>74099.400</X>
  <Y>5527.390</Y>
</Vertex>
<Vertex>
  <X>35852.300</X>
  <Y>41788.100</Y>
</Vertex>
<Vertex>
  <X>57104.000</X>
```

```
    <Y>85260.000</Y>
   </Vertex>
   <Vertex>
    <X>38919.500</X>
    <Y>199687.000</Y>
   </Vertex>
   <Vertex>
    <X>47789.800</X>
    <Y>258718.000</Y>
   </Vertex>
   <Vertex>
    <X>1745.580</X>
    <Y>261978.000</Y>
   </Vertex>
  </LandRegion>
```

The simple scenario needs only this one region, although it is possible to add a second one for the portion of Japan on the map.

Note: Strictly speaking, it is not necessary to define a land region because it has no objects that travel on land. However, this is a rare enough case that showing how to define a region is appropriate.

After the regions, the next step is to describe the *Decision Makers*. A Decision Maker is a role within the scenario that might be taken by a live player but need not be. The Decision Maker controls assets, and each asset must belong to a Decision Maker. A Decision Maker can control many assets, but each asset has only a single owning Decision Maker.

In the simple scenario, with two opposing assets, it is natural to define two Decision Makers. The full specification of a Decision Maker has the following four components:

- Role
- Identifier
- Color
- Briefing

The *role* is the position that this Decision Maker takes within the story of the game. Roles are used only for display purposes and so can be anything that makes sense for that particular game.

By contrast, the *identifier* is used in other places. For example, when an asset is specified as belonging to a particular Decision Maker, it is the identifier that is used to name the owner.

The color is a color that will be used to designate the assets belonging to this particular Decision Maker; the permissible color names are provided in Appendix 1. Note that while it is common to identify the two sides in many games as “Blue” and “Red,” those are just labels and have no necessary relationship to the display colors.

The optional *briefing* is a description of the job of this Decision Maker within the game. If used, the briefing can take any value that the experimenter chooses, as long as it contains no line breaks.

The sample scenario might contain the following:

```
<DecisionMaker>
  <Role>Blue Commander</Role>
  <Identifier>Player 1</Identifier>
  <Color>CornflowerBlue</Color>
  <Briefing>Find, identity and destroy all RedGroup units without loss of
civilian lives or property.</Briefing>
</DecisionMaker>
<DecisionMaker>
  <Role>Red Commander</Role>
  <Identifier>Player 2</Identifier>
  <Color>DarkRed</Color>
  <Briefing>Fly annoyingly over South Korea .</Briefing>
</DecisionMaker>
```

It is now possible to begin specifications of assets.

Sensor and Non-sense

Assets can have many features. This permits fine-grained definitions and extreme flexibility, but it requires quite a bit of specification for each asset. One way to reduce the load is to allow features that might belong to more than one asset to be defined separately and then referenced by name.

Sensors fall into that category of separable features. Before learning how to specify them, consider how they work. Sensor definition is quite complex, and the following information touches the surface of sensor definition.

A sensor is a form of asset that detects emissions of “signals” by other assets. Each sensor detects only one kind of signal, although there is a

“default” that makes it possible for one sensor to detect “everything.” If a player controls more than one sensor, then the view that asset has of other assets is the combined best view taken over all sensors. Players can belong to overlapping sensor networks. If two assets belong to the same network, then each has access to all sensor information collected by either asset.

Although the sample scenario has two non-cooperating players, it is necessary to define a sensor network for each player, the network names are for display purposes only. Each member is identified by its <Identifier> from the <Decision Maker> section of the scenario. In a more general case there might be many members listed for each network.

```
<Network>
  <Name>Buckeyes</Name>
  <Member>Player 2</Member>
</Network>
<Network>
  <Name>Wolverines</Name>
  <Member>Player 1</Member>
</Network>
```

The simplest way to specify a sensor is to give its name and range. This is a model in which the following is true:

- The sensor can detect all detectable attributes of other assets.
- The sensor has perfect detection within its range, and no detection at all beyond it.

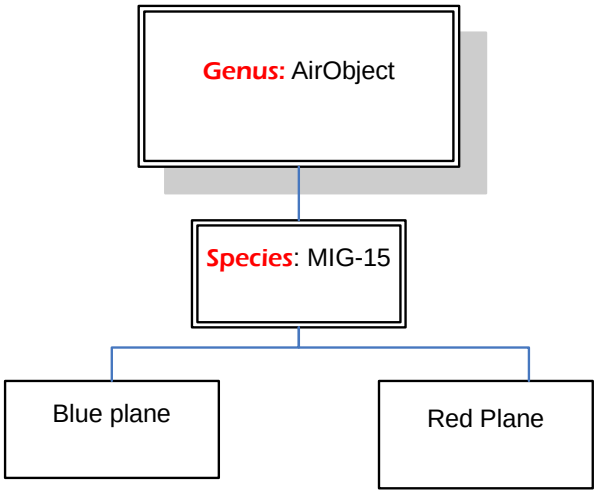
The following specifies the common radar systems of the two planes:

```
<Sensor>
  <Name>Radar</Name>
  <Extent>32000</Extent>
</Sensor>
```

Note that the tag “Extent” identifies the range.

Gentlefolk, start your engines

At last, it is time to define assets. Assets come at the leaves of a taxonomic tree that–with one exception–is defined entirely by the scenario writer. The following is a picture of that tree for the sample scenario.



The terms “genus” and “species” were chosen for their associations in the Linnaean taxonomic hierarchy. The following is true in each scenario:

- The genus gives the most basic categorization. The choices are LandObject, SeaObject, and AirObject.
- The species defines the characteristics of an object. Note that species can be derived from other species. If, for example, there were characteristics common to all MIGs and a scenario writer has more than one model of MIG in the scenario, the scenario writer can define a species called MIG and from it derive other species, such as MIG-15, and so on.
- The actual definition of an asset does little more than identify it. Other parts of the scenario instantiate the asset and give it things to do.

A genus, therefore, is not so much defined as referenced. It appears as follows:

```
<Genus>
  <Name>AirObject</Name>
</Genus>
```

Survival of the Species

The definition of a species begins with some handy identifying information. There are four items that might be supplied, although only two are required. The required items include:

- Name–this is the identifier that will be used to refer to this species.
- Base–this is the species or genus that is the immediate predecessor of this one in the taxonomy.

The optional items include:

- IsWeapon—set to true if this species defines a weapon; defaults to false.
- RemoveOnDestruction—if set to true, assets of this species will disappear when they are destroyed; if set to false, which is the default, they will remain on the screen when destroyed.

```
<Species>
  <Name>Mig-15</Name>
  <Base>AirObject</Base>
  <IsWeapon>false</IsWeapon>
  <RemoveOnDestruction>true</RemoveOnDestruction>
```

Confederated states

The behavior and characteristics of assets belonging to a particular species at any one time are determined by its state. A *state* is a named collection of attribute values. Every asset is "in" one of its states at all times, and various external actions can cause it to change state.

For an example outside of the sample scenario, an asset might change states when damaged by a weapon. In its new state, it might have a lesser fuel capacity or fewer offensive capabilities. Perhaps, though, if it returns to base for repairs, it can return to its original state. As its state changes, so might the icon used to represent it, as well as its sensing abilities and the signals that it emits.

Every species must define a state called FullyFunctional. The species may define other states as well. Every species must also have a state called Dead. There is no need to explicitly define a Dead state for any species. It will be provided by default. Scenario writers might want to explicitly define a Dead state in a few special cases, such as to provide a different icon. On the other hand, the MaximumSpeed attribute of the Dead state will always be forced to 0, as shown in a later example.

The definition of the FullyFunctional state, which must precede any other state definitions, goes between the following brackets:

```
<FullyFunctional>
</FullyFunctional>
```

The attribute values that go between those brackets are described in the next section. Any other state definition, all of which follow FullyFunctional, has the following overall structure:

```
<DefineState>
  <State>nameOfState</State>
</DefineState>
```

With one exception, the possible elements in the definition of FullyFunctional are the same as for other states. That exception is the icon used to represent the asset on the screen. An icon definition is *required* for FullyFunctional but is optional for other states.

This, incidentally, is an indicator of a very powerful characteristic of states. States can be inherited both by longitude and latitude. That is, when one species is defined based on another, it automatically inherits the state definitions of its parent, although it may override any of them. Also, when a species has more than one state, all of the attribute values of FullyFunctional are inherited by other states, unless they choose to supply their own values. The one exception to this is the Dead state, which can do nothing of interest no matter what attribute values it is given.

For Attribution

What then are the attributes that can be defined for a state? After the icon, they can be conceptually organized into four separate groups: parameter values, senses, capabilities and vulnerabilities. All of these are optional. After the icon, all of these parameters are enclosed in the following pair of brackets:

```
<StateParameters>
</StateParameters>
```

Parameter values are specifications of basic characteristics of the asset in that state. Beyond the icon, which has been mentioned, there are seven parameter values. The sample scenario will use the following five parameter values:

- MaximumSpeed: Upper limit on speed in meters/second
- FuelCapacity: How much fuel the asset can carry (in liters)
- InitialFuelLoad: How much fuel the asset first has (in liters)
- FuelConsumptionRate: how quickly the asset burns fuel (in liters/meter)
- FuelDepletionState: The state to which the asset transitions when it runs out of fuel

For the MIG-15 the following values can be used:

```
<MaximumSpeed>298.61</MaximumSpeed>
<FuelCapacity>1400</FuelCapacity>
<InitialFuelLoad>1400</InitialFuelLoad>
<FuelConsumptionRate>.00117</FuelConsumptionRate>
<FuelDepletionState>Dead</FuelDepletionState>
```

The second group of attributes–Senses–lists the sensors on the asset. In our case there is only one possible sensor, shown below:

```
<Sense>Radar</Sense>
```

Next in order are the capabilities and vulnerabilities. These are a story unto themselves.

If you tickle us, do we not laugh?

Capabilities and vulnerabilities fit together like keys into locks. Only when one unit's capabilities match another's vulnerabilities can the first affect the other. The basic notion is that capabilities produce (unitless) numbers, and vulnerabilities define state changes based on those numbers. For a simple example of how this works, imagine three units called A, B, and C. With the proper definitions, it might be the case that A can destroy B (meaning, push it into the Dead state). This could have less catastrophic effects or no effect at all on C.

In general, an asset can have many capabilities. These can have effects depending on distance and can combine effects with other assets. For now, the sample scenario will use the following simple capability:

```
<Capability>
  <Name>Destroy</Name>
  <Proximity>
    <Radius>2000</Radius>
    <Effect>
      <Intensity>100</Intensity>
      <Probability>0.7</Probability>
    </Effect>
  </Proximity>
</Capability>
```

This segment begins with a name. Every capability has a name. Names are arbitrary but allow the scenario writer to mimic the situation where a unit has two or more different ways to affect others. Each capability then has a

collection of distance-related effects. There is just one effect in this sample scenario. This effect specifies that the capability can be used when the asset is within two kilometers of another unit. If the player chooses to employ this capability in this situation, there is a 70% chance that the capability will be effective. The intensity of its effect is 100. This intensity value is crucial to making capabilities (and vulnerabilities) work together.

The value of 100 is, as noted, arbitrary. But, in conjunction to similar numbers seen in the definition of vulnerabilities, it simulates an amount of "force" being applied to another asset. If the sample scenario had many species, for some of them, 100 would be enough to push them into the Dead state. For others, the same intensity might only disable them, and the same value might have no effect at all on a third asset. What happens to an asset when a capability is employed against it is defined by its vulnerabilities? Here is an example:

```
<SingletonVulnerability>
  <Capability>Destroy</Capability>
  <Transitions>
   <Effect>99</Effect>
   <State>Dead</State>
  </Transitions>
 </SingletonVulnerability>
```

The reason for the term *Singleton*Vulnerability will be explained later in this guide. This segment tells the DDD that this asset has a vulnerability to the Destroy capability. An asset with that capability might be able to affect it. There are only two choices for what happens. Nothing at all will happen if the intensity of the deployed capability is less than 99. In other cases, this asset will move into the Dead state. Since the capability is defined as having intensity 100, scenario writers can see that 70% of the time that one of these assets attacks the other from within two kilometers, the attack will be a complete success.

Can you hear me? Can you feel me near you?

The last item in the specification of a species describes the signals it emits. These are the same signals that might be detected by a sensor. The sample scenario uses the same simple "Default" specification to indicate that it is potentially possible to detect everything there is to know about an asset, as shown below.

```
<Emitter>
 <Attribute>Default</Attribute>
```

```
    </Emitter>
```

More precisely, this specifies that whether some other asset can detect this one depends only on the qualities of its sensors. It is possible to specify that the signal emitted drops off with distance, or is masked with Gaussian noise.

This completes the definition for the species.

Putting It All Into Play

So far, all the sample scenario has is definitions, but nothing is happening. For that last step, it is necessary to tell the DDD that there are some assets. For example,

```
 <Create_Event>
  <ID>BlueMig</ID>
  <Kind>Mig-15</Kind>
  <Owner>Player 1</Owner>
</Create_Event>
```

describes the Blue plane, and

```
 <Create_Event>
  <ID>RedMig</ID>
  <Kind>Mig-15</Kind>
  <Owner>Player 2</Owner>
</Create_Event>
```

defines the Red plane.

A <Create_Event> tag does not in itself bring an asset into play. For that, a <Reveal_Event> tag is required. In its simplest form, the <Reveal_Event> brings an asset into play at a specified time, in a specified location. For example,

```
 <Reveal_Event>
  <ID>BlueMig</ID>
  <Time>1</Time>
  <InitialLocation>
   <X>1060.00</X>
   <Y>431525</Y>
   <Z>0</Z>
  </InitialLocation>
 </Reveal>
```

puts the blue plane in play at once, and

```
<Reveal_Event>
  <ID>RedMig</ID>
  <Time>1</Time>
  <InitialLocation>
    <X>44226</X>
    <Y>901675</Y>
    <Z>0</Z>
  </InitialLocation>
</Reveal_Event>
```

places the red plane over Pyongyang.

The Reveal event has an optional InitialState parameter, which specifies the initial state of the object, referring to the states that were defined in the reverent Species definition. If not specified, the InitialState will be "FullyFunctional."

Ready, set, …

The Blue Plane is to be controlled by a live player in the sample game, so nothing more need be done for it. The Red Plane must get its directions from the script. First, it should fly to Seoul. For example,

```
<Move_Event>
   <ID>RedMig</ID>
   <Timer>1</Timer>
   <Throttle>100</Throttle>
   <Position>
     <X>141383</X>
     <Y>727617</Y>
   </Position>
 </Move_Event>
```

causes the Red Plane to fly to Seoul at full throttle. When it arrives, it should then fly back to Pyongyang. This requires using a Completion_Event. It specifies that when the last movement for RedMig is complete, there is something else to do:

```
<Completion_Event>
   <ID>RedMig</ID>
```

```
  <Action>Move_Complete</Action>
  <DoThis>
   <Move_Event>
    <ID>RedMig</ID>
    <Timer>25</Timer>
    <Throttle>95</Throttle>
    <Position>
     <X>44226</X>
     <Y>901675</Y>
    </Position>
   </Move_Event>
  </DoThis>
 </Completion_Event>
```

In order to have the Red Plane fly back to Seoul and then back to Pyongyang, simply add more Completion events, as shown below.

```
<Completion_Event>
  <ID>RedMig</ID>
  <Action>Move_Complete</Action>
  <DoThis>
 <Move_Event>
  <ID>RedMig</ID>
  <Timer>1</Timer>
  <Throttle>100</Throttle>
  <Position>
   <X>141383</X>
   <Y>727617</Y>
  </Position>
 </Move_Event>
  </DoThis>

 </Completion_Event>
<Completion_Event>
  <ID>RedMig</ID>
  <Action>Move_Complete</Action>
  <DoThis>
    <Move_Event>
     <ID>RedMig</ID>
    <Timer>25</Timer>
    <Throttle>95</Throttle>
```

```
    <Position>
   <X>44226</X>
   <Y>901675</Y>
    </Position>
   </Move_Event>
  </DoThis>
</Completion_Event>
```

Chapter 3: Regions

Flat and Tall

The term region is used to define both a two-dimensional outline on land ands a three-dimentional volume of space.

Land regions

A *land region* is a two-dimentional outline that describes the boundaries of land. It defines an area that a land-based object can travel on. If two land regions are adjacent then a land-based object can move from one to the other. A scenario may have many land regions.

Each land region is defined by its polygonal boundary. There is an example in the Chapter 2 scenario that we already developed.

Any portion of a playfield not in a land region is perforce a sea region. In our Chapter 2 scenario, by not defining a land mass to contain Japan, we made it permeable to submarines and impossible for planes to land on. If an asset attempts to travel where it should not – a land-based object moving into the sea, a frigate coming onto dry land, it is destroyed.

Scoring regions

A scoring region is a prism – that is, it is a three dimensional object with uniform a polygonal cross section. Like a land region, a scoring region has an identifier and a list of vertices. It's lowest value on the z-axis is called its <Start> and its highest value is called its <End>. Either or both of these may be negative, as a scoring region may extend below land. So a scoring region might be defined like this:

```
<ScoringRegion>
```

```
  <ID>Hill 17</ID>
  <Vertex>
   <X>191221</X>
   <Y>12541800</Y>
  </Vertex>
  <Vertex>
   <X>341950</X>
   <Y>7482430</Y>
  </Vertex>
  <Vertex>
   <X>366491</X>
   <Y>4956400</Y>
  </Vertex>
  <Start>0</Start>
  <End>760</End>
 </ScoringRegion>
```

The primary purpose of a scoring region is to define a place within the simulation where a player accumulates (or loses) points. The mechanism for this is TBD.

Active regions

An active region represents a three dimensional space that in some way affects the operation of the game. In addition to the <ID>, <Vertex> list and <Start> and <End> values, an active region defines these values:

- SpeedMultiplier – the speed of any asset inside this region is multiplied by the value; it is probably less than one, but need not be.
- BlocksMovement – has the value *true* if assets are not capable of moving through this space; the value is *false* otherwise
- SensorBlocked – *true* if sensors are blocked by this space.

Chapter 4: Teams and Networks--Decision makers and their groupings

Teams

There are two ways to group players in a scenario. The more fundamental of the two is the team. A *team* is a functional alliance of players. Typically,

teams in scenarios define allies, with the precise meaning of "ally" of course depending on the roles that the scenario assigns to players.

The team definitions follow the definitions of regions[1]. For each team there is a name, and then a list of other teams that it is hostile to, defined with the tag <Against>. Here's a sample

```
<Team>
  <Name>Eastern Alliance</Name>
  <Against>Northern Coalition</Against>
  <Against>Delian League</Against>
</Team>
```

An asset controlled by a player can attack any other asset, including its own teammates; thus friendly fire can be a problem in DDD games. What the team specifications determine is what happens when a controlled asset passes near one that is running autonomously, under control of the scenario. An autonomously controlled asset will routinely attack assets belonging to teams they are hostile to as a self-defense action.

No player can belong to more than one team. Furthermore, the relationship between teams need not be symmetric; thus, for example, it could be the case that Northern Coalition is hostile to Delian League, but that Delian League is not hostile to Northern Coalition.

Team Membership

Membership in a team is indicated in the <DecisionMaker> command. We've already seen many of the elements of this command; here's one instance from Chapter 2:

```
  <DecisionMaker>
    <Role>Blue Commander</Role>
    <Identifier>Player 1</Identifier>
    <Color>CornflowerBlue</Color>
    <Briefing>Find, identity and destroy all RedGroup units without loss of civilian lives or property.</Briefing>
    <Team>BlueTeam</Team>
```

[1] See Appendix 2 for the order of the elements of a scenario.

```
</DecisionMaker>
```

Networks

Networks are also groupings of players. There are likely to be some correlation between the teams and the networks, but this is not required. The Network lists come after the definitions of the Decision Makers: each contains the name of the network and its members. There were examples of networks with one member in the first scenario; an example with two members follows:

```
<Network>
  <Name>Wolverines</Name>
  <Member>Player 1</Member>
 <Member>ReservePlayer4</Member>
</Network>
```

Networks are the fundamental units of sensor fusion. *Sensor fusion* describes the combining of information from different sensors. Specifically, in the DDD, a player has access to all of the sensor information that is available to any member of any network to which the player belongs. It is not very likely that a player would share network membership with someone in an antagonistic team, but there could be a number of cooperating teams, and thus there could be networks with members from more than one team. A player may belong to many networks.

Chapter 5: Sensors and emitters

Sensors

Earlier, we saw an example of a simple sensor:

```
<Sensor>
  <Name>Radar</Name>
  <Extent>32000</Extent>
</Sensor>
```

This is a sensor that looks in all directions and detects anything that is within its range of 32 km. Sensors in the DDD can be defined to be much more specific.

A sensor has a name and is defined in terms of three characteristics: the attribute, the cone, and the extent. The *attribute* is a value that the sensor detects from other assets; our simple example detected all possible attributes, so none needed to be specified.

The field in which a sensor detects objects is a *cone*, with apex at the sensor. To specify a cone one provides these four elements:

- Spread: the *spread* is the angle of the cone's apex
- Extent: as before, the *extent* is the distance to which the sensor's abilities extend
- Direction: the direction is a three-dimensional vector, which, together with the location of the cone, specifies the direction of the cone's axis.
- Level: the level specifies how well the sensor works within this cone. We'll return to discuss this important notion below.

Here is what the definition of a cone might look like; this particular cone spans 60 degrees, points into the sky at a 45 degree angle, and detects the sensor's attribute within 1000 m: The value for "level" is A; this will become clearer when we discuss emitters, below.

```
<Cone>
   <Spread>60</Spread>
   <Extent>1000</Extent>'
   <Direction>
     <X>1</X>
     <Y>0</Y>
     <Z>1</Z>
   </Direction>
   <Level>A</Level>
</Cone>
```

One sensor may have many cones. This reflects the dropping off of the signal's detection capabilities with distance. Thus, the most straightforward – and possibly the most realistic – situation would give a sensor a number of cones, all with the same direction, but with increasing spread and extent; larger spread and extent would correspond to a lower level of signal detection.

Emitters

Emitters are properties of individual assets. Although the term seems to imply activity, in this context any signal that can be detected is defined by an emitter; among these would be size, position, and remaining fuel. The typical signal attenuates over distance. Therefore, like the definition of a cone, the definition of an emitter must reflect this effect of distance.

We've seen a simple emitter, one that is the analogue of the simple sensor. It was described by

```
<Emitter>
  <Attribute>Default</Attribute>
</Emitter>
```

indicating that any of its attributes that *might* be sensed *can* be sensed. Other emitters may be limited to particular signals. For example, this emitter definition shows that the size of the asset can be detected by the right sensors:

```
<Emitter>
  <Attribute>Size</Attribute>
  <Level>A</Level>
  <Variance>20</Variance>
</Emitter>
```

The other values – the <Level> and the <Variance> describe characteristics of the signal. The values of <Level> are arbitrary, but are intended to be consonant with the <Level> values used in sensor definitions. The <Variance> describes the accuracy with which the signal can be detected.

The detection accuracy of a numeric quantity assumes that the error in a signal is normally distributed with the actual value as the mean and the variance as given by the <Variance> value.

Let's see how this works in this case; for convenience, let s represent the square root of the <Variance> value – 4.47 in this case.. Suppose the actual size of the asset were 10m. The DDD will choose a value (z-score) from the normal distribution N(0,1) with mean 0 and standard deviation 1. Suppose that its (uniform) random number generator produces the value .56. This corresponds to a z-score of 0.15. That value is transformed by the reverse z-transform to the distribution N(10,4.47) to produce the value

$$(4.47)(.56)+10=12.50$$

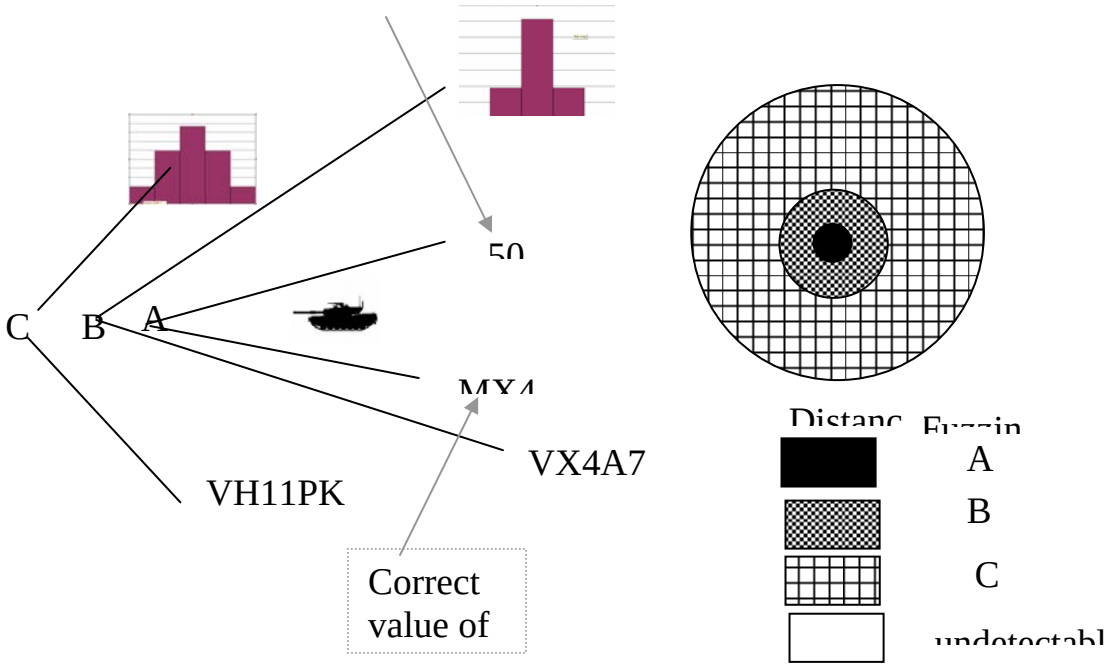
That is to say, a sensor that detected the size of assets at level “A” would, in this instance, read the size of this particular object as 12.5. Of course, since the readings are probabilistic, another reading would most likely yield a different value; over time, the readings would be normally distributed around the true value, 10, with standard deviation 4.47.

Since the <Level> ultimately determines how precisely the signal is read, it may be helpful to use the term *fuzziness levels*. In the DDD it is possible for a sensor to detect text strings that may represent attributes of an asset, such as the name of the team that its owning player belongs to. In that case, the <Variance> value is replaced by a <Probability>. If the value of <Probability> is p, then each individual character of the string to be detected is replaced with probability p; if it is replaced, the new character will be chosen from all displayable characters according to a uniform distribution.

Just as a sensor might detect a signal at different levels, perhaps varying with distance, so too an asset might emit the same signal at different levels. Notice that while the cones of a sensor need not be nested, the fuzziness levels do impose an ordering of the levels. The following picture may help to clarify the relationship between sensors and emitters.

On the left there is a tank and, although they are not centered on it (for legibility), also given are the fuzziness definitions for two attributes, one numeric (50) and one a string (MX417W). Fuzziness level C gives 100% accurate rendition (so the <Variance> for one attribute is 0, as is the <Probability> for the other), but readings at levels B and A will be perturbed by errors. On the top on this side are schematic representations of the shapes of the probability distributions for the numeric value at each of the two fuzziness levels; the two text strings “VXA7W” and “VH11PK” show how the string might be garbled at each level.

On the right the Figure shows cross sections of the cones for a sensor: the outermost circle achieves detection with fuzziness level C, the next with B, and the innermost with A – outside the third level there is no detection at all. This diagram is simplified in this way: each of the attributes will actually have been detected by a separate sensor, and there is no particular reason why the two sensors have to have the same radial rings.



An emitter with three different levels would be defined like this, where we deliberately use non-parallel names for the levels to reinforce the point that the level names are arbitrary codes.

```
<Emitter>
  <Attribute>Size</Attribute>
  <Level>A</Level>
  <Variance>20</Variance>
  <Level>W3</Level>
  <Variance>16</Variance>
  <Level>203.5</Level>
  <Variance>27</Variance>
</Emitter>
```

Chapter 6: Capabilities and vulnerabilities

Now we understand how sensors allow the assets that you don't control be displayed on your screen. Now you might like to do something to those assets. There are two mechanisms that you can use. They both have the same root concepts. We'll look at those concepts, and one of the methods here.

Capabilities

Every asset can potentially have the ability to affect other objects: a fighter plane might have a laser weapon such as the 100-kilowatt infrared laser, which is being developed for the F35 Joint Strike Fighter. Or a truck might be able to load fuel onto a tank. Or a medical team might be able to vaccinate someone against a disease. We call these capabilities.

More precisely, a capability is a way that one asset can cause a change of state in another asset. We saw a capability defined in Chapter 2:

```
<Capability>
 <Name>Destroy</Name>
 <Proximity>
  <Radius>2000</Radius>
  <Effect>
   <Intensity>100</Intensity>
   <Probability>0.7</Probability>
  </Effect>
 </Proximity>
</Capability>
```

A capability is simply a name (like the "Level" we saw in sensors and emitters). For the examples we mentioned above, we might have used names like "Laser" or "Vaccinate" or "Refuel" – but any strings would be equally acceptable.

After the name of the capability is a <Proximity> group. The tag name is motivated by the idea that the group tells what happens to assets within a specified distance of the unit when the capability is deployed. That's, of course, the <Radius> -- as usual in Astral, the units are meters. Specifically, a <Proximity> group is applied if the distance between the asset and the target is *less than or equal to* the <Radius>.

What the capability does to an asset is an *effect*. An effect is defined by its intensity and its probability. The *probability* is the probability that, if the assets are within range, the effect will actually happen; for example, aiming a laser might be more precise or take less time at a short distance than at a long one, and so there would be a higher probability of the laser being successful at a shorter range.

You might suspect that there could be a number of different proximity groups for one capability. You would be correct. Here's an example of a capability with three different levels of effect. The probability of being affected by an electromagnetic pulse (EMP) is 1 for any device within range, but the intensity drops off with distance.

```
<Capability>
 <Name>ElectroMagneticPulse</Name>
 <Proximity>
  <Radius>16240</Radius>
  <Effect>
   <Intensity>100</Intensity>
   <Probability>1</Probability>
  </Effect>
 </Proximity>
 <Proximity>
  <Radius>32480</Radius>
  <Effect>
   <Intensity>50</Intensity>
   <Probability>1</Probability>
  </Effect>
 </Proximity>
 <Proximity>
  <Radius>48720</Radius>
  <Effect>
   <Intensity>33</Intensity>
   <Probability>1</Probability>
  </Effect>
 </Proximity>
 <Proximity>
  <Radius>64980</Radius>
  <Effect>
   <Intensity>25</Intensity>
   <Probability>1</Probability>
  </Effect>
 </Proximity>
</Capability>
```

In contrast, here is a capability that is equally effective within the ranges shown, but which has an accuracy that decreases with distance:

```
<Capability>
  <Name>.308 Winchester Cartridge</Name>
  <Proximity>
    <Radius>185</Radius>
    <Effect>
      <Intensity>100</Intensity>
      <Probability>1</Probability>
    </Effect>
  </Proximity>
  <Proximity>
    <Radius>278</Radius>
    <Effect>
      <Intensity>100</Intensity>
      <Probability>.92</Probability>
    </Effect>
  </Proximity>
  <Proximity>
    <Radius>370</Radius>
    <Effect>
      <Intensity>100</Intensity>
      <Probability>76</Probability>
    </Effect>
  </Proximity>
  <Proximity>
    <Radius>463</Radius>
    <Effect>
      <Intensity>100</Intensity>
      <Probability>.61</Probability>
    </Effect>
  </Proximity>
  <Proximity>
    <Radius>649</Radius>
    <Effect>
      <Intensity>100</Intensity>
      <Probability>.31</Probability>
    </Effect>
  </Proximity>
</Capability>
```

One could also have a capability where both intensity and probability drop off with distance.

We have yet to explain <Intensity>. Although <Intensity> is numeric, it is like <Level> in having no intrinsic meaning. What it means for an asset to be affected by a capability at some intensity level depends entirely on the particular asset's vulnerabilities.

Vulnerabilities

The vulnerabilities of an asset describe its changes of state as a result of the application of a capability. The vulnerability we saw defined in Chapter 2 was the following:

```
<SingletonVulnerability>
 <Capability>Destroy</Capability>
 <Transitions>
  <Effect>99</Effect>
  <State>Dead</State>
 </Transitions>
</SingletonVulnerability>
```

The <Capability> tag indicates that this asset can be affected by some other asset that has the capability "Destroy." The transition defines the intensity level necessary to cause the unit to change state. Here we see that if the intensity is *greater than or equal to* 99, the value of <Effect>, the state transition will occur.

Again, there can be different results of an attack for different levels of effect. The following describes a person wearing hypothetical body armor with varying levels of vulnerability:

```
<SingletonVulnerability>
 <Capability>.308 Winchester Cartridge</Capability>
 <Transitions>
  <Effect>199</Effect>
  <State>Dead</State>
 </Transitions>
 <Transitions>
  <Effect>170</Effect>
  <State>OutOfAction</State>
 </Transitions>
```

```
    <Transitions>
      <Effect>80</Effect>
      <State>PartlyEffective</State>
    </Transitions>
  </SingletonVulnerability>
```

A unit may have many vulnerabilities, each to a different capability.

Combined attacks

Given the examples in the last two sections it might seem that the person wearing the body armor can at worst be made partly effective (i.e., wounded) by a .308 Winchester cartridge; any other state change requires more effect than the cartridge can deliver at any distance.

However, two or more of the cartridges could have a *combined* intensity of 200. in the DDD it is possible for two different assets to deploy the same capability within a short period of time. In this case the individual intensities are calculated and summed to determine whether a state change occurs in the target.

Combining effects from the same capability applied by different assets is only one of the ways in which cooperative attacks can be required by the scenario. Another kind of combining comes through what is called a ComboVulnerability; note that it is the existence of a ComboVulnerability that requires the use of the name SingletonVulnerability.

When an asset has a ComboVulnerability it can be affected (induced to change state) by a combination of different capabilities. When a vulnerability depends on a combination of capabilities, we do not expect that the effect numbers of the different capabilities will be commensurate. Rather, the scenario writer describes the combination of effect values that will cause the state change. For example, to indicate that an asset changes state to “Dead” as a result of near simultaneous application of capability A at intensity level 10 or higher and capability B at intensity 35 or higher:

```
<ComboVulnerability>
  <Contribution>
    <Capability>A</Capability>
    <Effect>10</Effect>
  </Contribution>
```

```
  <Contribution>
    <Capability>B</Capability>
    <Effect>35></Effect>
  </Capability>
  <New State>Dead</New State>
</ComboVulnerability>
```

An asset may have many singleton and combo vulnerabilities – or none.

Chapter 7: States

Most of the attributes of assets are determined by their states. This includes
Capabilities
Vulnerabilities
Emitters
Sensors:
and some parameters that we'll discuss in this chapter.

States, behaviors, and inheritance

The logic of tying so much of the identity of an asset to its states is simple: it allows the asset's abilities to change during the play of the game. Although many games may need no more than the two required states – FullyFunctional and Dead – by defining additional states the scenario writer can create almost unlimited behavioral richness. An asset might lose some of its abilities because of one state change, and regain some or all from another. Each state can offer a unique combination. It is even possible for an asset to gain abilities beyond those of its "FullyFunctional" state.

This may seem contradictory, but it follows from the unique role of the FullyFunctional state as it interacts with inheritance of attributes. Two forms of inheritance govern the attributes of asset. One is related to the hierarchy of species. When one species is based on another, the child species begins with the same state definitions as the parent. Any definitions made for the child override those for the parent; thus, a child need not have a FullyFunctional state defined at all; it will inherit its parent's.

However, in addition to parent-child inheritance there is also inheritance from the FullyFunctional state to other states of the same asset. Another

interesting wrinkle is that some attributes are inherited differently from others.

The process works like this. We begin with the FullyFunctional state of the parent (if there is one). The attribute values in this state are overridden by any values in the child, on a one-by-one basis, with a few exceptions. If a child has no sensors defined then it inherits those of its parent – but if it has any sensors defined, it inherits none from its parent. This is not the case for emitters: inheritance there is on an attribute-by-attribute basis.

Then the other states of the child, with the exception of the Dead state, inherit from Fully Functional; any values that are not explicitly defined for that state.

Other attributes of states:

Other attributes also determine the behavior of an asset in a given state. They are the following:

- MaximumSpeed: The fastest speed attainable by the asset
- FuelCapacity: The amount of fuel that the asset can carry in this state
- InitialFuelLoad: The amount of fuel that the asset starts with in this state
- FuelConsumptionRate: The rate at which fuel is consumed when the asset is moving at its maximum velocity
- FuelDepletionState: The state the asset is put into when it runs out of fuel

A few additional attributes refer to characteristics that have not yet been defined.

Each of these values has a default. Maximum speed defaults to 0; such an asset cannot move. By contrast, if the fuel amounts and rate are not provided, they are not taken into account during the game. If the depletion state is not defined, it is the Dead state.

Chapter 8: Weapons and subplatforms

Assets in the DDD can carry other objects: planes carry weapons, an aircraft carrier carries planes, and the planes may carry pilots who have handguns, and so on. This chapter addresses the definition and meaning of such relationships.

Subplatforms

A *subplatform* is an asset carried by another asset. Subplatforms have the same owning decision maker as the host asset does, but there are no other required relationships between a subplatform and its host. Thus, a land object may carry sea objects and assets without capabilities can carry very capable subplatforms.

Actually, to speak of "carrying" a subplatform is somewhat misleading. An asset that is physically distant from another can still be a subplatform. Thus, the planes from an aircraft carrier remain subplatforms even after they are launched. When a subplatform is "on" its host, we say that it is *docked*. In some cases, subplatforms may begin the game in an undocked state, and may dock and undock throughout the game. The time it takes an asset to dock with its parent is a definable asset of a state of the subplatform, called the DockingDuration. Similarly, the time it takes for a launch of a subplatform to be effective is the LaunchDuration.

When a subplatform is docked, it can be destroyed only when its host is. However, once it has been launched its existence is independent of that of its host.

Weapons

Weapons are a limited class of subplatform. The limits are these:

- They cannot begin the game undocked
- When launched, they are directed at a specific target
- Their path cannot be altered once launched.

Defining Subplatforms

Subplatforms are defined like any other asset. Define a species, or a hierarchy of species, to have the characteristics of the subplatform.

Bringing them into existence is done differently. Here's a piece of scenario that would be part of a <Create_Event> for a host asset.

```
<Subplatform>
 <Kind>AGM-114</Kind>
 <Docked>
  <Count>2</Count>
 </Docked>
</Subplatform>
```

It states that the host has two AGM-114 type subplatforms, that begin docked. These docked subplatforms can be given initial parameters – all must have the same initial parameters – like this

```
<Subplatform>
 <Kind>AGM-114</Kind>
 <Docked>
  <Count>2</Count>
  <InitialParameters>
   <MaximumSpeed>1800</MaximumSpeed>
 </Docked>
</Subplatform>
```

A subplatform that is a weapon is so declared with the <IsWeapon> tag of the <Species> command.

Subplatforms themselves may carry weapons. This is declared right after the <Kind> tag. The following example states that every subplatform in this grouping carries five P-14S weapons. If there were other subplatforms of type AGM-114 that carried different weapon complements, they would be defined in a different <Armament> statement.

```
<Subplatform>
 <Kind>AGM-114</Kind>
 <Armament>
  <Weapon>P-14S</Weapon>
  <Count>5</Count>
  </Armament>
 <Docked>
  <Count>2</Count>
  <InitialParameters>
   <MaximumSpeed>1800</MaximumSpeed>
```

```
  </Docked>
 </Subplatform>
```

As noted above, some of the subplatforms may begin launched. Launched subplatforms are declared right under the <Docked> ones: for each one, between <Launched></Launched> brackets, it is necessary to provide an initial location, an initial state (defaulting to FullyFunctional) and optional initial parameters. When there are more than one with the same initial parameter settings they can be declared by giving the starting location of each, between one pair of <Launched></Launched> brackets, as in the following excerpt:

```
  <Launched>
    <Location>
      <X>4680</X>
      <Y>5040</Y>
      <Z>0</Z>
    </Location>
    <Location>
      <X>9610</X>
      <Y>1110</Y>
      <Z>0</Z>
    </Location>
    <InitialState>FullyFunctional</InitialState>
  </Launched>
```

Appendix 1: Color Names:

AliceBlue
Aqua
Azure

Blue
BlueViolet
Brown

CornflowerBlue
Crimson
Cyan

DarkBlue
DarkGray
DarkGreen
DarkKhaki
DarkOliveGreen
DarkRed
DarkSlateGray
DodgerBlue

Fuchsia

Gray
Green
GreenYellow

HotPink

Khaki

Lavender
LimeGreen

Magenta
Maroon

Navy

Olive
OliveDrab
Orange
OrangeRed

Plum
PowderBlue
Purple

Red
RosyBrown
RoyalBlue

SandyBrown
SeaGreen
Silver
SkyBlue
SlateBlue
SlateGray

Tan
Teal
Transparent
Turquoise

Violet

Yellow
YellowGreen

Appendix 2: Order of items in a scenario

Note: this chart sacrifies detail for clarity. The final arbiter for the form of a scenario is the schema file.

Name (of scenario)
Description
Playfield
Regions
Teams
Decision Makers
Networks
Sensors
Events

Copyright © 2006 Aptima, Inc. All rights reserved.