

APTIMA[®]
HUMAN-CENTERED
ENGINEERING

DDD v4.0 Scenario Writing Guide

Revision 2

March 23, 2007

Contents

User Guide	2
Chapter 1: Introduction	3
What is the DDD?	3
The Skinny on Scenarios	3
XML—The Scenario Writer’s Swiss Army Knife	3
The Life-Cycle of a Scenario	4
What Next?	5
Chapter 2: Overview of a Simple Scenario	6
Telling a Story	6
The Structure of Scenarios	7
Scaffolding	8
The Outermost Shell	8
The Region.....	9
The Decision Makers	11
Sensor and Non-sense	12
Gentlefolk, Start Your Engines	13
Survival of the Species.....	14
Confederated States	14
For Attribution	15
Parameter Values	15
Senses	15
Capabilities and Vulnerabilities: If You Tickle Us, Do We Not Laugh?	16
Emitters: Can You Hear Me? Can You Feel Me Near You?.....	17
Putting It All Into Play	17
Ready, Set,	18
Chapter 3: Regions	21
Flat and Tall	21

Land Regions	21
Active Regions	21
Chapter 4: Teams and Networks—Decision Makers and Their Groupings	22
Teams	22
Team Membership	22
Networks	22
Chapter 5: Sensors and Emitters	24
Sensors	24
Emitters	25
Chapter 6: Capabilities and Vulnerabilities	27
Capabilities	27
Effects and Parameters	27
Vulnerabilities	29
Combined Attacks	30
Chapter 7: States	32
States, Behaviors, and Inheritance	32
Other Attributes of States	32
Chapter 8: Weapons and Subplatforms	34
Subplatforms	34
Weapons	34
Defining Subplatforms	34
Chapter 9: Scores	37
Existence Rules	37
Encounter Rules	37
The Actor Template	38
Appendix 1: Color Names	40
Appendix 2: Order of Items in a Scenario	41
Reference Guide	42
Tag Index	43
Diagrams	52

User Guide

Chapter 1: Introduction

This document describes 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>
  </Zip>
</Locale>
</Address>

```

The items within angle 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>`”), 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 that scenario 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. All commands and the most important options are presented in the second section of this manual: the Reference Guide that begins on page 43.

Chapter 2: Overview of a Simple Scenario

This chapter explains what a scenario is by describing a simple example.

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 Universal Transverse Mercator (UTM) coordinates. The UTM Coordinate system provides coordinates on a flat grid of a world map, which is divided into 60 zones. Each zone has a width of 6 degrees longitude. *Easting* and *northing* are geographic Cartesian coordinates for any given point. Easting is the *x* coordinate, meaning, the measured distance toward the East. Northing is the *y* coordinate, meaning, the measured distance toward the North. In the UTM Coordinate system, northing is the distance to the equator, and easting is the distance to a location in the UTM zone.

Thus, we will provide a UTM grid identifier. In this case, that identifier is S-52, 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.

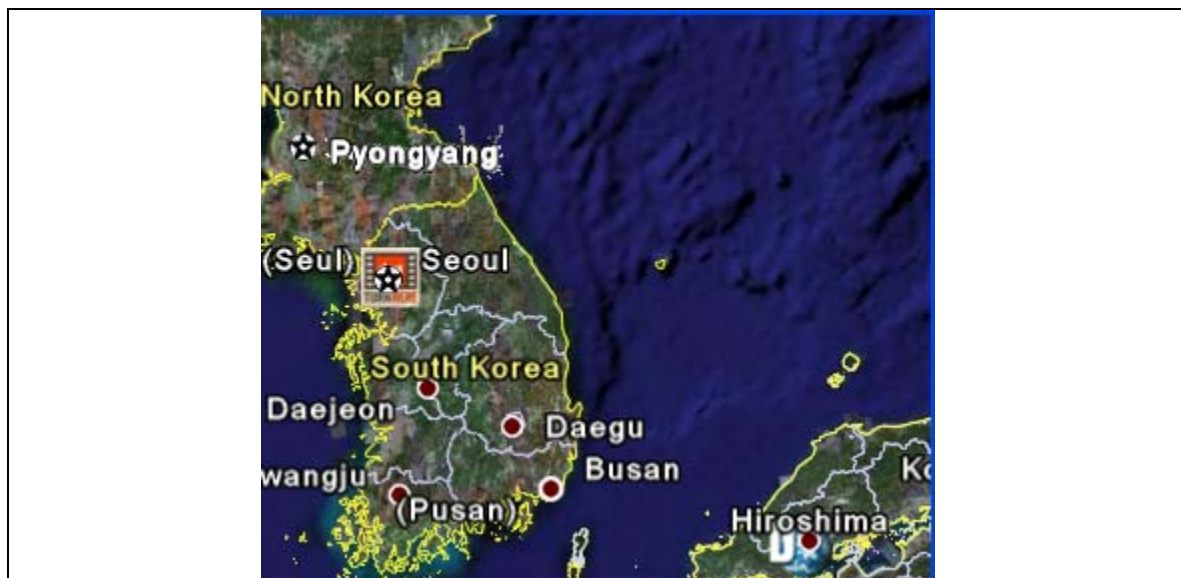


Figure 1. The playing field

In addition, 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. 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, as shown below.

<Scenario>

Contents of
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. In addition, it is necessary to provide information about the field of play, as shown below.

<Scenario>

<ScenarioName>...</ScenarioName>

<Description>...</Description>

<Playfield>

Describe
playfield

</Playfield>

Remainder

</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. They 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>

Describe
playfield

</Playfield>

Describe
Decision
Makers

Describe
assets

Specify
actions

</Scenario>

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, which include:

- The outermost shell
- The region
- The Decision Makers
- Sensors

The Outermost Shell

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.

This specification goes after the `</Description>` and before the `</Scenario>`, so the complete scenario thus far looks 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>154.47</ VerticalScaling >
    <HorizontalScaling>117</ HorizontalScaling >
  </Playfield>
</Scenario>
```

The Region

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.

Figure 2, for example, shows the boundary to be used for the Korean peninsula.



Figure 2. The boundary 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>6418, 16012100</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>1305.050, 492731.000</Vertex>
<Vertex>131555.000, 495060.000</Vertex>
<Vertex>79998.600, 444510.000</Vertex>
<Vertex>86778.900, 421986.000</Vertex>
<Vertex>83456.700, 397680.000</Vertex>
<Vertex>167983.000, 234975.000</Vertex>
<Vertex>179371.000, 151747.000</Vertex>
<Vertex>167152.000, 82291.600</Vertex>
<Vertex>131300.000, 40488.100</Vertex>
<Vertex>74099.400, 5527.390</Vertex>
<Vertex>35852.300, 41788.100</Vertex>
<Vertex>57104.000, 85260.000</Vertex>
<Vertex>38919.500, 199687.000</Vertex>
<Vertex>47789.800, 258718.000</Vertex>
<Vertex>1745.580, 261978.000</Vertex>
</LandRegion>
```

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

The Decision Makers

After the regions, the next step is to describe the *Decision Makers*. A Decision Maker is a role within the scenario that might, but need not, be taken by a live player. 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 this simple scenario, with two opposing assets, we 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,” these are just labels. There is no direct relationship between the Decision Maker’s name and its display color. For example, if the scenario writer creates a Decision Maker called Red, it is a common practice to assign the color red to that player, but this is not required. It is also common to have Decision Makers that are not named after 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 requires quite a bit of specification for each asset. One way to reduce the load is by allowing 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 only touches its surface.

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. 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. Figure 3 is a picture of that tree for the sample scenario.

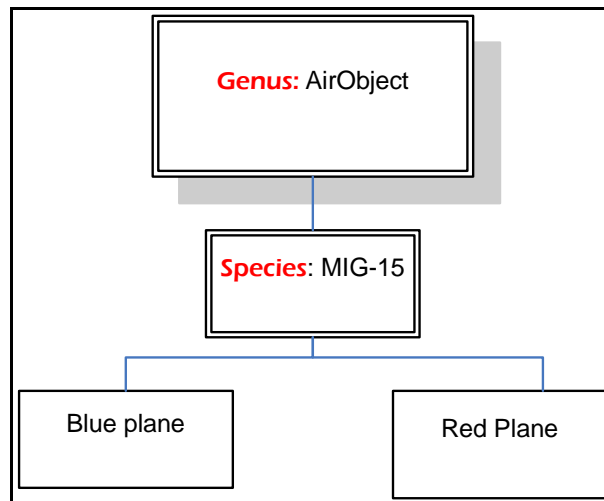


Figure 3. A taxonomic tree defined by the scenario writer

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 species 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 (which is the default), assets of this species will disappear when they are destroyed. If set to false, these assets will remain on the screen when destroyed.

An example follows:

```
<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 an asset 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 longitudinally and latitudinally. 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. In addition, when a species has more than one state, the required attribute values of the states are inherited by FullyFunctional, unless the definitions of the states 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 five separate groups:

- Parameter values
- Senses
- Capabilities
- Vulnerabilities
- Emitters

All of these are optional. After the icon, all of these parameters are enclosed in the following pair of brackets:

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

Parameter Values

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>
```

Senses

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.

Capabilities and Vulnerabilities: 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 capabilities 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. However, in conjunction with similar numbers seen in the definition of vulnerabilities, this value simulates an amount of “force” (in arbitrary units) being applied to another asset. If the sample scenario had many species, 100 would be enough to push some of 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 group.

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 *SingletonVulnerability* 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 when one of these assets attacks the other from within two kilometers, the attack will be a complete success.

Definitions of vulnerabilities, like those of capabilities, can have ranges and probabilities as part of their definitions. This is a more complete definition of a singleton vulnerability:

```
<SingletonVulnerability>
  <Capability>FIRE</Capability>
  <Transitions>
    <Effect>1</Effect>
    <Range>30</Range>
    <Probability>.5</Probability>
    <State>Casualties</State>
    <Effect>2</Effect>
    <Range>50</Range>
    <Probability>.7</Probability>
    <State>Casualties</State>
  </Transitions>
</SingletonVulnerability>
```

Here it takes a larger effect to cause a state change from 50 meters than from 30 meters. However, the probability of success is higher at 50 meters than at 30 meters.

Emitters: 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 and in a specified location. For example,

```
<Reveal_Event>
  <ID>BlueMig</ID>
  <Time>1</Time>
  <InitialLocation>1060.00, 431525, 0</InitialLocation>
</Reveal>
```

puts the Blue plane in play at once, and

```
<Reveal_Event>
  <ID>RedMig</ID>
  <Time>1</Time>
```

```
<InitialLocation>44226, 901675, 0</InitialLocation>
</Reveal_Event>
```

initially 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 relevant 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>141383, 727617</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. In this example, the Completion_Event 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>44226, 901675</Position>
    </Move_Event>
  </DoThis>
</Completion_Event>
```

The timer value is taken to be a relative time from the completion of the previous 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>141383, 727617</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>44226, 901675</Position>
    </Move_Event>
  </DoThis>
</Completion_Event>
```

There is another way to get this same effect and more. A Reiterate command allows for infinite repetition of a series of moves. The movement of the Red plane could be expressed this way:

```
<Reiterate>
  <Start>1</Start>
  <ReiterateThis>
    <Move_Event>
      <ID>RedMig</ID>
      <Timer>25</Timer>
      <Throttle>95</Throttle>
      <Position>44226, 901675</Position>
    </Move_Event>
    <Move_Event>
      <ID>RedMig</ID>
      <Timer>1</Timer>
      <Throttle>100</Throttle>
      <Position>141383, 727617</Position>
    </Move_Event>
  </ReiterateThis>
</Reiterate>
```

As each of these commands completes, the next one is queued up for performance at the indicated (relative) time. That command is in effect then moved to the bottom of the ReiterateThis list. This plane will keep flying back and forth until it runs out of fuel, is attacked, or is made to do something else by its owning Decision Maker.

Chapter 3: Regions

This chapter describes the different kinds of regions in a scenario.

Flat and Tall

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

Land Regions

A *land region* is a two-dimensional outline that describes the boundaries of land. It defines an area on which a land-based object can travel. 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 landmass to contain Japan, we made it permeable to submarines and on which it is impossible for planes to land. If an asset attempts to travel where it should not—for example, a land-based object moving into the sea or a frigate coming onto dry land—that asset is destroyed.

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` is 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` is *true* if sensors are blocked by this space.

Chapter 4: Teams and Networks—Decision Makers and Their Groupings

This chapter describes how to organize players in the scenario.

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.

In the scenario file, the team definitions follow the definitions of regions. (See Appendix 2 for the order of the elements of a scenario.) For each team there is a name and then a list of other teams to which it is hostile, defined with the tag `<Against>`. Here is 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 occur and have consequences in DDD games. The team specifications determine what happens when a controlled asset passes near an asset that is running autonomously, under control of the scenario. An autonomous asset—one not controlled by a live Decision Maker—will routinely attack assets belonging to teams to which they are hostile as a self-defense action.

No player can belong to more than one team. Furthermore, the relationship between teams need not be symmetric. For example, it could be the case that Northern Coalition is hostile to Delian League, but 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 is one instance from Chapter 2:

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

Networks

Networks are also groupings of players. There is 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 are 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

This chapter provides an in-depth look at sensors and emitters.

Sensors

Earlier, we saw the following 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 the following three characteristics:

- The attribute
- The cone
- 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, it is necessary to provide these four elements:

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

Here is what the definition of a cone might look like.

```
<Cone>
  <Spread>60</Spread>
  <Extent>1000</Extent>
  <Direction>1, 0, 1</Direction>
  <Level>A</Level>
</Cone>
```

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, which will become clearer when we discuss emitters, below.

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.

Consider how this works in this case. For convenience, let s represent the square root of the **<Variance>** value, which is 4.47 in this example. 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 following 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 to which its owning player belongs. 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. Figure 4 may help to clarify the relationship between sensors and emitters.

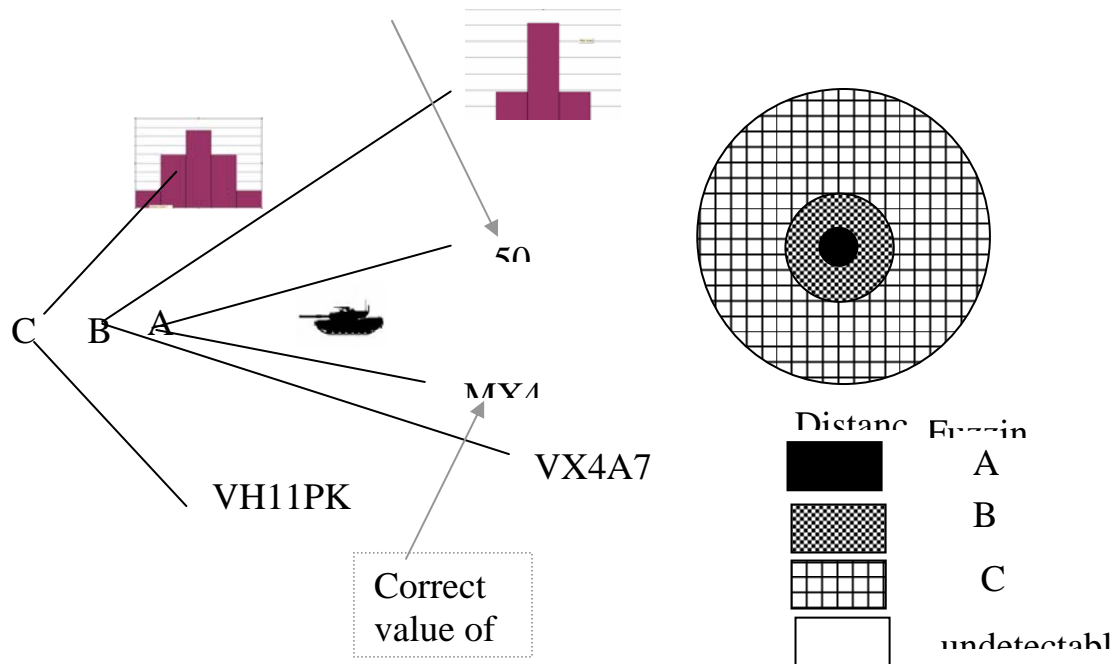


Figure 4. The relationship between sensors and emitters

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 A 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 C 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, Figure 4 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 actually will 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 that we understand how sensors allow the assets that you do not control to be displayed on your screen, 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 will look at those concepts, and one of the methods here.

Capabilities

Every asset can potentially have the ability to affect other objects. For example, 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 the following 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. The distance, of course, is 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>*.

Effects and Parameters

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 is 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 16 km, 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.

Recall that vulnerabilities can also be modified by probabilities and ranges.

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 (for example, 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*.

When an asset has a ComboVulnerability it can be affected (meaning, 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.

The following example indicates 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>  
  </Contribution>  
  <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
- Other parameters, some of which are described 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 FullyFunctional state.

However, in addition to parent-child inheritance, there is 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 a child has any sensors defined, it inherits none from its parent. This is not the case for emitters. Inheritance for emitters 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:

- LaunchDuration is the length of time required to complete the operation of launching a subplatform.
- DockingDuration is the length of time required to complete the operation of docking a subplatform.
- TimeToAttack is the length of time that an attack is active. During that time other attacks can add to the initial attack.
- MaximumSpeed is the fastest speed attainable by the asset.
- FuelCapacity is the amount of fuel that the asset can carry in this state.
- InitialFuelLoad is the initial amount of fuel that the asset has in this state.
- FuelConsumptionRate is the rate at which fuel is consumed when the asset is moving at its maximum velocity.
- FuelDepletionState is the state the asset is put into when it runs out of fuel.

Each of these values has a default. Maximum speed defaults to 0, meaning, the 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. For example, planes carry weapons, an aircraft carrier carries planes, and the planes may carry pilots who have handguns. 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, 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 destroyed. However, once a subplatform has been launched, its existence is independent of that of its host.

Weapons

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

- Weapons cannot begin the game undocked.
- When launched, weapons are directed at a specific target.
- The path of a weapon cannot be altered once that weapon is 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 subplatforms into existence is done differently. The following is a piece of a scenario that would be part of a `<Create_Event>` for a host asset.

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

The preceding example states that the host has two AGM-114 type subplatforms, which 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 subplatform, between `<Launched></Launched>` brackets, it is necessary to provide an initial location, an initial state (defaulting to FullyFunctional), and optional initial parameters. When there is more than one subplatform with the same initial parameter settings, these subplatforms can be declared by giving the starting location of each subplatform, between one pair of `<Launched></Launched>` brackets, as shown in the following excerpt:

```
<Launched>
  </Z>
  </Location>
  <Location>9610, 1110, 0 </Location>
  <InitialState>FullyFunctional</InitialState>
</Launched>
```

As shown above, if only two components are given for a location or position, then the third (Z) defaults to 0.

Chapter 9: Scores

A *score* is best thought of as an accumulator or counter. Certain events—which are defined in *rules*—can cause changes to the contents of the score. The rules can be limited by giving partial or complete specifications of the units involved in the event. A given rule can belong to many different scores.

Each score specifies one or more Decision Makers, indicated in the scenario by the tag “Applies.” When a rule is evaluated, as described below, it is evaluated for each score to which it belongs and for each “applicable” Decision Maker specified by the score.

There are two kinds of rules:

- Existence rules
- Encounter rules

Existence Rules

An *existence rule* is evaluated for every unit at the end of every time cycle. Think of these as seconds. The rule contains, among other items, an “actor” definition. Think of an actor as being a template with two places to fill in. One of these places is filled in by a Decision Maker, and the other is filled in by a unit. For example, an actor definition might say, in effect, “Unit 1 belongs to a Decision Maker who is hostile to Decision Maker A.” For each live unit, the rule is evaluated by seeing if the actor definition is true when the unit in question is plugged into the first position and an applicable Decision Maker is plugged into the second.

The components of an existence rule are shown in the following table:

Field	Typical values	Default	Meaning
Name	any string	None	Identifies the rule for later reference.
Unit	an actor	None	This rule is only triggered if focus matches this actor definition.
Increment	number	None	Delta to score when this rule applies.

Encounter Rules

An *encounter rule* is evaluated whenever one unit applies a capability to another. In this case there are two actor definitions. Suppose Unit X is applying a capability to unit T. The rule is evaluated with X as the first actor and T as the second, and also with T as the first actor and X as the second. Again, both of these evaluations will be done for every Decision Maker applicable to a score and for every score.

The components of an encounter rule are shown in the following table:

Field	Typical values	Default	Meaning
Name	any string	none	Identifies the rule for later reference.
Unit	an actor	none	This rule is only triggered if unit initiating encounter (using a capability) matches this actor definition.
Object	an actor	none	This rule is only triggered if unit being encountered (through a vulnerability) matches this actor definition.
NewState	name of a state	none	The rule is triggered if the Object transitions to this state as a result of the encounter.
From	Name of a state	""	The rule is triggered if the Object began the encounter in this state; "" means rule is not limited by initial state.
Increment	number	none	Delta to score when this rule applies.

The Actor Template

The components of an actor template are shown in the following table:

Field	Typical values	Default	Meaning
Owner	"Hostile", "Friendly", "This"	none	Tells the relation between the unit involved and the focus.
ID	"All", species name, unit name	none	Identifies the unit or type of unit that is involved
Location	A location definition	(Anywhere, InZone)	Designates places in which the unit must be for the rule to apply.

The actor can specify the following three classes of owner for the unit plugged into the actor:

- A unit hostile to the applicable Decision Maker
- A unit friendly to the applicable Decision Maker
- A unit that belongs to that Decision Maker

In addition, the unit being plugged in must match the ID field, which can specify an actual unit name, the name of a species, or all units. Notice too that the rule may only be active in certain parts of the playfield. The **<Location>** field names one or more active regions, and determines whether the rule applies for units inside the regions or outside of them. The components of a region definition are shown in the following table:

Field	Typical values	Default	Meaning
Zone	“Anywhere” or list of regions	None	Regions to which rule pertains, modified by Relationship
Relationship	“InZone,” “NotInZone”	“InZone”	Actor’s relationship to region

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 list sacrifices detail for clarity. The final arbiter for the form of a scenario is the schema file.

In general, scenario items should be arranged in the following order:

1. Name (of scenario)
2. Description
3. Playfield
4. Regions
5. Teams
6. Decision Makers
7. Networks
8. Sensors
9. Events

Reference Guide

Tag Index

Tag	Appears on page	Within	Purpose
ScenarioName	ScenarioName		Name used to identify scenario
Description	ScenarioName		A text description of the scenario; this will be displayed to all players
Land Region	Playfield		The boundaries of a land area
ID	Playfield	LandRegion	Name for the region; referred to by scoring rules
Vertex	Playfield	LandRegion	One of the points on the boundary of the region; the region is expected to be convex.
ActiveRegion	Playfield		A three-dimensional space with properties affecting unit movement, visibility, or scoring
ID	Playfield	ActiveRegion	Name for the region; referred to by scoring rules
Vertex	Playfield	ActiveRegion	One of the points on the boundary of the region; the region is expected to be convex.
Start	Playfield	ActiveRegion	Lowest vertical(z) level for region; may be negative
End	Playfield	ActiveRegion	Highest vertical(z) level for region; may be negative
SpeedMultiplier	Playfield	ActiveRegion	Factor (typically <1.0) applied to speed of any object passing through region
BlocksMovement	Playfield	ActiveRegion	Tells whether units may move through region
SensorBlocked	Playfield	ActiveRegion	List of sensors that this region blocks
Name	Team		Unique name for team of players
Against	Team		List of teams to which this team is hostile
Role	DecisionMaker		Description of the role taken by this player in the scenario

Tag	Appears on page	Within	Purpose
Identifier	DecisionMaker		Unique name used to identify player on the screen
Color	DecisionMaker		Unique name for color used to identify this player's assets
Briefing	DecisionMaker		A mission briefing displayed to this player
Team	DecisionMaker		Team to which player belongs
Name	Network		Unique name for a network of players that share sensor data
Member	Network		List of network members
Name	Sensor		Unique name for this sensor
Attribute	Sensor		Attribute detected by sensor
Cone	Sensor		Region of detection of sensor
Extent	Sensor		Radius of sphere of detection
Spread	Sensor	Cone	Apex angle of cone
Extent	Sensor	Cone	Distance from apex in which cone is active
Direction	Sensor	Cone	Vector direction of axis
Level	Sensor	Cone	Arbitrary code that matches with emitters
TimeToAttack	(self)		Default duration after initiation of any attack on this unit
Genus	Genus/Species		Unique name of genus: "LandRegion", "SeaRegion" or "AirRegion"
Species	Genus/Species		Describes the species
Name	Genus/Species	Species	Unique name of the species
Base	Genus/Species	Species	Species or genus from which this one directly inherits
Size	Genus/Species	Species	Radius of assets of this type (for collision detection)

Tag	Appears on page	Within	Purpose
IsWeapon	Genus/Species	Species	Whether these units are weapons
RemoveOnDestruction	Genus/Species	Species	Whether units of this type are removed from screen when destroyed
<FullyFunctional> ¹	Genus/Species	Species	Definition of this required state
<DefineState>	Genus/Species	Species	Definitions of other states
Name	Capability		Unique name of the capability
Proximity	Capability		A possible way to use this capability
Range	Capability	Proximity	Distance from unit that this is operative
Effect	Capability	Proximity	What the capability does when used
Intensity	Capability	Effect	Size of effect (in arbitrary units)
Probability	Capability	Effect	Chance that an application of this characteristic of the capability will succeed
Icon	DefineState and FullyFunctional	FullyFunctional only	Default image to be used for assets of this species
State	DefineState	DefineState only	Unique name for state
Icon	DefineState	DefineState only	Image to be used for assets of this species when they are in this state
StateParameters	DefineState		Settable characteristics of the state
LaunchDuration	DefineState		Length of time required to launch a unit of this species
DockingDuration	DefineState		Length of time required to dock a unit of this species
TimeToAttack	DefineState		Overrides global TimeToAttack for this state
MaximumSpeed	DefineState		Maximum speed of asset when not in active zone
FuelCapacity	DefineState		Amount of fuel unit can store

¹ Angle brackets indicate that the components of this item are found on a different page.

Tag	Appears on page	Within	Purpose
InitialFuelLoad	DefineState		Amount of fuel available to unit when entering this state
FuelConsumptionRate	DefineState		Rate at which fuel is used up in this state
FuelDepletionState	DefineState		New state after fuel supply exhausted
Sense	DefineState		List of sensors available to unit in this state
<Capability>	DefineState		Description on an attack capability
<SingletonVulnerability>	DefineState		Description of a vulnerability to a single capability
<ComboVulnerability>	DefineState		Description of vulnerability to a combination of capabilities
<Emitter>	DefineState		Description of a characteristic that may be detectable by sensors
Capability	SingletonVulnerability		Name for a capability to which this unit is vulnerable
Transitions	SingletonVulnerability		Possible effects of application of this capability
Effect	SingletonVulnerability	Transitions	“Amount” of capability required to invoke this vulnerability
Range	SingletonVulnerability	Transitions	Distance within which capability must be applied
Probability	SingletonVulnerability	Transitions	Probability that an application of this capability will cause a state change
State	SingletonVulnerability	Transitions	State to which unit transitions after capability is applied
Contribution	ComboVulnerability		Describes what one of the cooperating attackers must do to invoke this vulnerability
Capability	ComboVulnerability	Contribution	Name of capability that must be applied
Range	ComboVulnerability	Contribution	Distance within which

Tag	Appears on page	Within	Purpose
			capability must be applied
Effect	ComboVulnerability	Contribution	Level of effect that must be applied
Probability	ComboVulnerability	Contribution	Probability that this contribution will be considered—that the application of the capability will be successful—as part of the cooperative application of capabilities
NewState	ComboVulnerability	Contribution	New state if all contributions are considered
Attribute	Emitter		Name of detectable signal
Level	Emitter		Arbitrary code that matches with sensors
Percent	Emitter		Describes application of noise to a text attribute
Variance	Emitter		Describes normal distribution, centered on true value, used to simulate noise in a numeric signal
ID	Create_Event		Name of unit being created
Kind			Species on which unit is based
Owner			Owning DecisionMaker
Subplatform			Description of a (family of) platform with which this unit is equipped
Kind		Subplatform	Species of subplatform
Armament			A class or weapon on the unit
Setting	InitialParameters, StartupParameters	Parameters	Value to assign to attribute
<EngramRange>	Reveal_Event		Conditions based on engrams that must be satisfied for Reveal to occur
Time	Reveal_Event		Time unit is to be revealed
InitialLocation	Reveal_Event		Position of unit when revealed

Tag	Appears on page	Within	Purpose
InitialState	Reveal_Event		Starting state for unit
<StateParameters>	Reveal_Event		Attribute assignments made when unit appears
Name	EngramRange		Unique name for memory variable
Included	EngramRange		List of values; variable must take one of them for success
Excluded	EngramRange		List of values; success if variable takes none of them
ID	Move_Event		Name of unit being moved
<EngramRange>	Move_Event		Conditions based on engrams that must be satisfied for Move to occur
Timer	Move_Event		Time at which move occurs
Throttle	Move_Event		Initial setting for unit throttle (as percent of full throttle) when movement begins
Destination	Move_Event		Three-dimensional position to which unit moves
ID	Completion_Event		Unit being monitored
<EngramRange>	Completion_Event		Conditions based on engrams that must be satisfied for Completion_Event to be triggered
NewState	Completion_Event		Transition to this state triggers new event
Action	Completion_Event		Notice of one of these actions triggers event ²
ID	StateChangeEvent		Unit to undergo change of state
<EngramRange>	StateChangeEvent		Conditions based on engrams that must be satisfied for state change to be triggered
Time	StateChangeEvent		Time at which state changes

² Possible events are Move_Complete, StateChangeNotice, SelfDefenseAttackNotice (that this unit began self-defense attack).

Tag	Appears on page	Within	Purpose
NewState	StateChangeEvent		State changed to
From	StateChangeEvent		Unit must have been in one of these states for change to be effective
Except	StateChangeEvent		Unit may not be in any of these states for unit to be effective
ID	Transfer_Event		Unit being transferred between players
<EngramRange>	Transfer_Event		Conditions based on engrams that must be satisfied for transfer to be triggered
Time	Transfer_Event		Time of transfer
From	Transfer_Event		Player losing unit
To	Transfer_Event		Player gaining unit
Parent	Launch_Event		Unit from which subplatform is to be launched
<EngramRange>	Launch_Event		Conditions based on engrams that must be satisfied for launch to be triggered
Time	Launch_Event		Time of launch
Kind	Launch_Event		Species of subplatform to be launched
Child	Launch_Event		Identifier of individual platform to be launched
RelativeLocation	Launch_Event		Vector added to parent's position to determine initial position of subplatform
InitialState	Launch_Event		Starting state for subplatform
<StartupParameters>	Launch_Event		Attribute values assigned to subplatform upon launch
Name	DefineEngram		Variable being created
Value	DefineEngram		Initial value for variable
Name	ChangeEngram		Variable being changed
Time	ChangeEngram		Time of change
Value	ChangeEngram		New value
Name	RemoveEngram		Variable being deleted

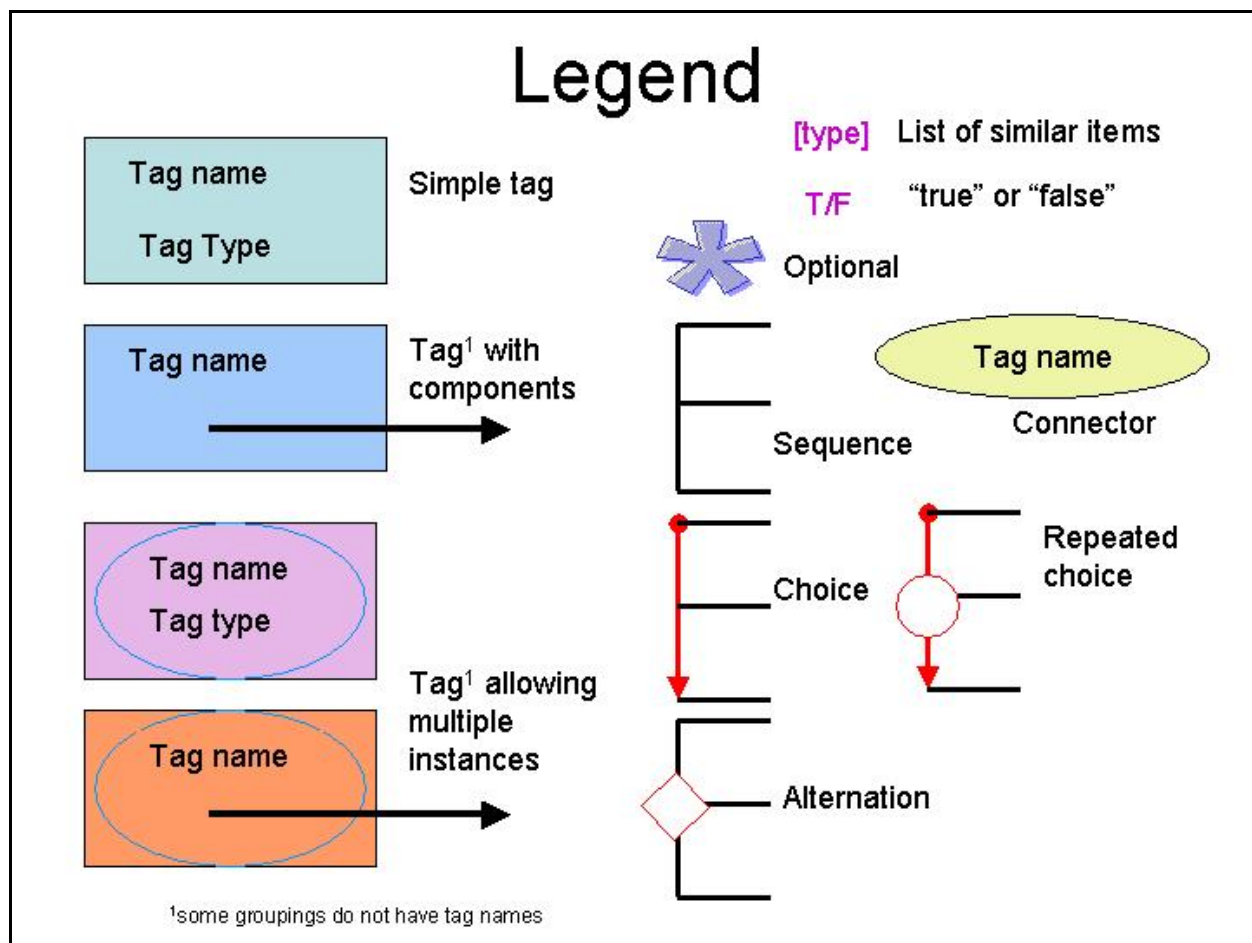
Tag	Appears on page	Within	Purpose
Time	RemoveEngram		Time of deletion
Unit	FlushEvents		Asset whose events are to be flushed
Time	FlushEvents		Time for events to be flushed
Start	FlushEvents		Time at which event cycle is to begin
<EngramRange>	FlushEvents		Conditions based on engrams that must be satisfied for launch to be triggered
ReiterateThis	FlushEvents		List of events to cycle ³
Room	OpenChatroom		Name of new chatroom
Time	OpenChatroom		Time to create chatroom
Members	OpenChatroom		List of members of chatroom
Room	CloseChatroom		Name of room to be closed
Time	CloseChatroom		Time to close room
Name	Rule		Name assigned to this scoring rule
Unit	Rule		Relationship of owner of primary asset in rule to owner named in rule ⁴
Object	Rule		Target unit in rule
Owner	Rule	Unit, Object	Owning player
ID	Rule	Unit, Object	Unit or species name
Zone	Rule	Unit, Object	List of active regions in which rule applies
Relationship	Rule	Unit, Object	Location of asset relative to active regions
NewState	Rule		Transition of object into this state after engagement triggers rule
From	Rule		Object must have previously been in this state
Increment	Rule		Amount by which any score using this rule is modified

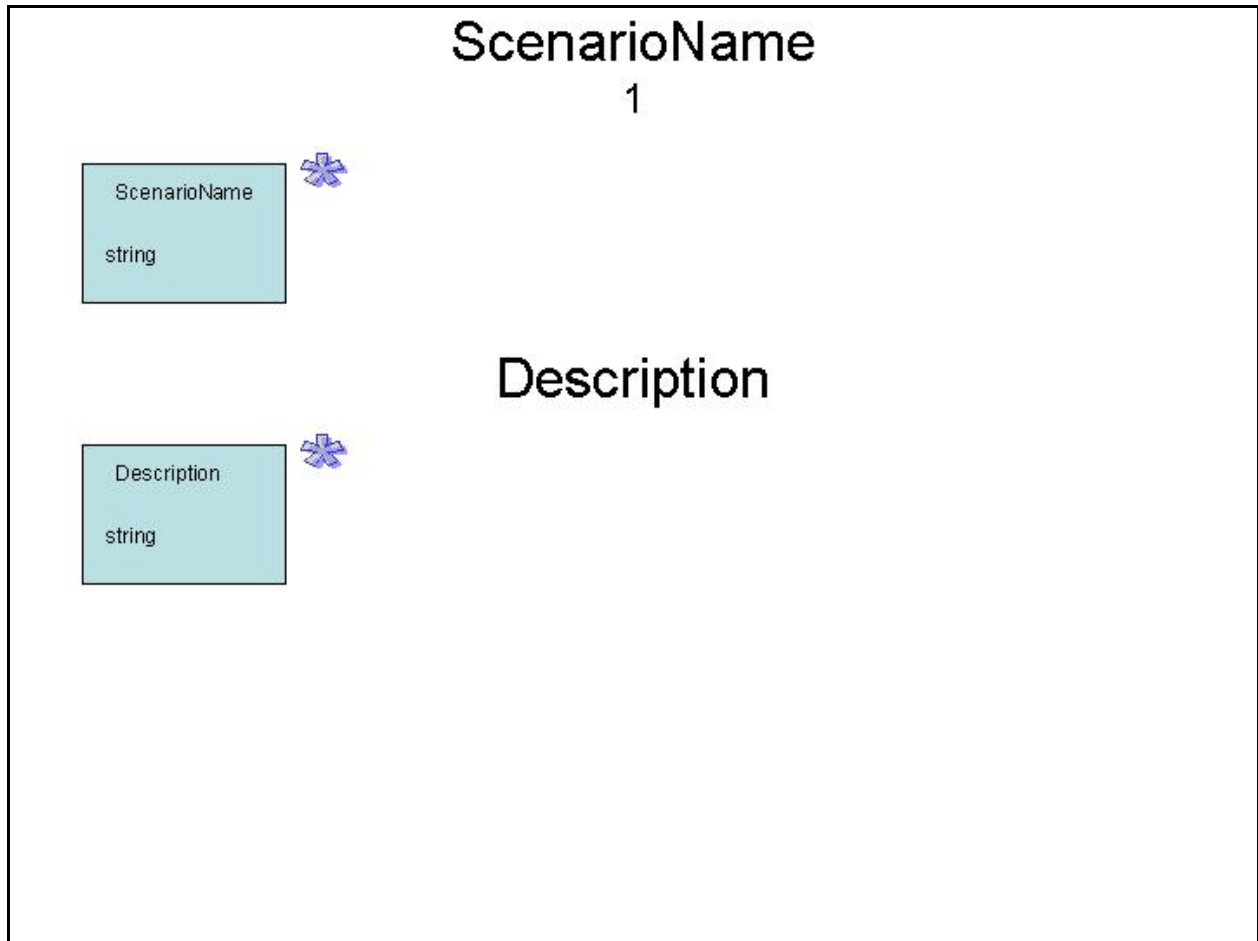
³ Only Move_Events based on Move_Complete events may be placed in a cycle.

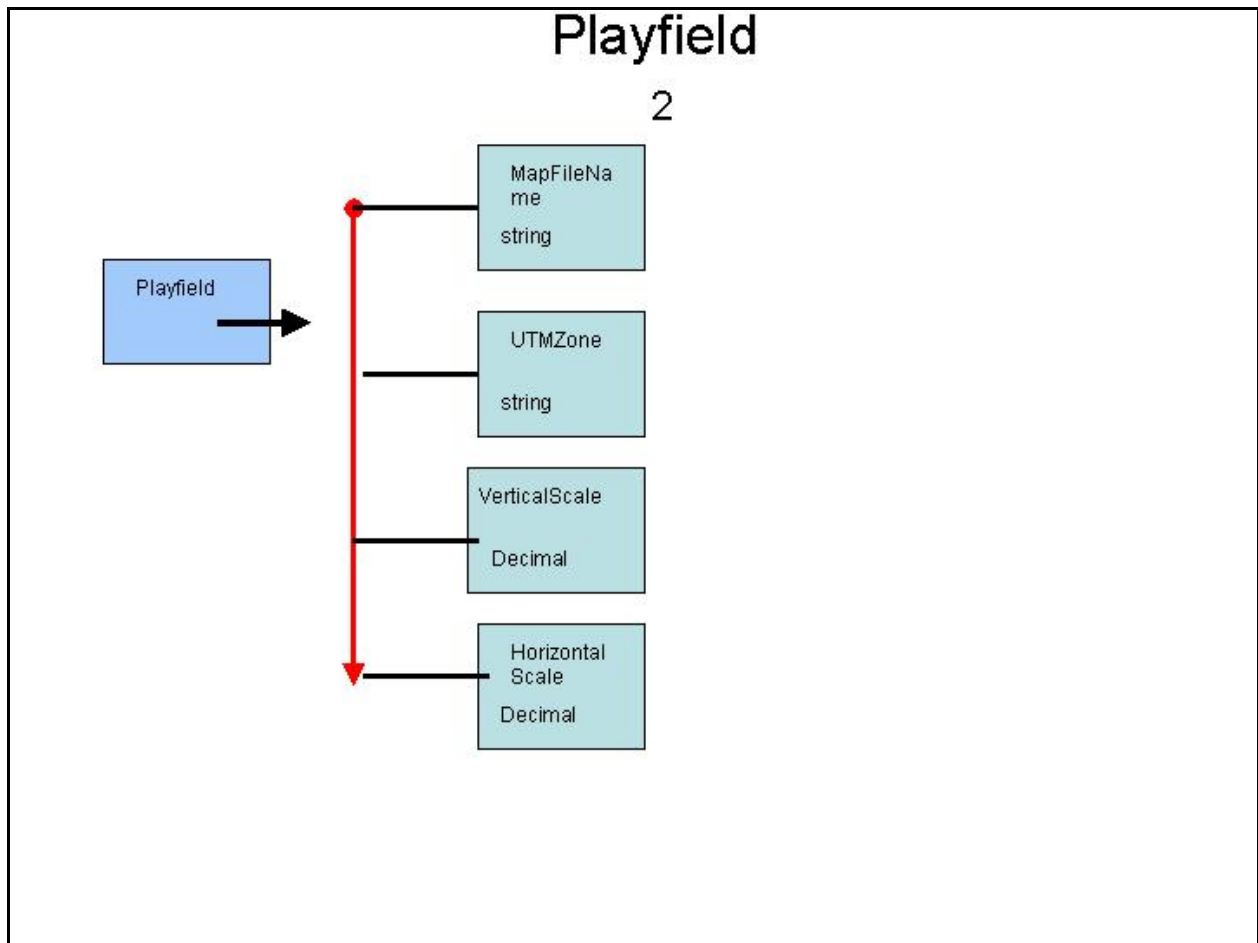
⁴ Possible values are Friendly, Hostile, or This.

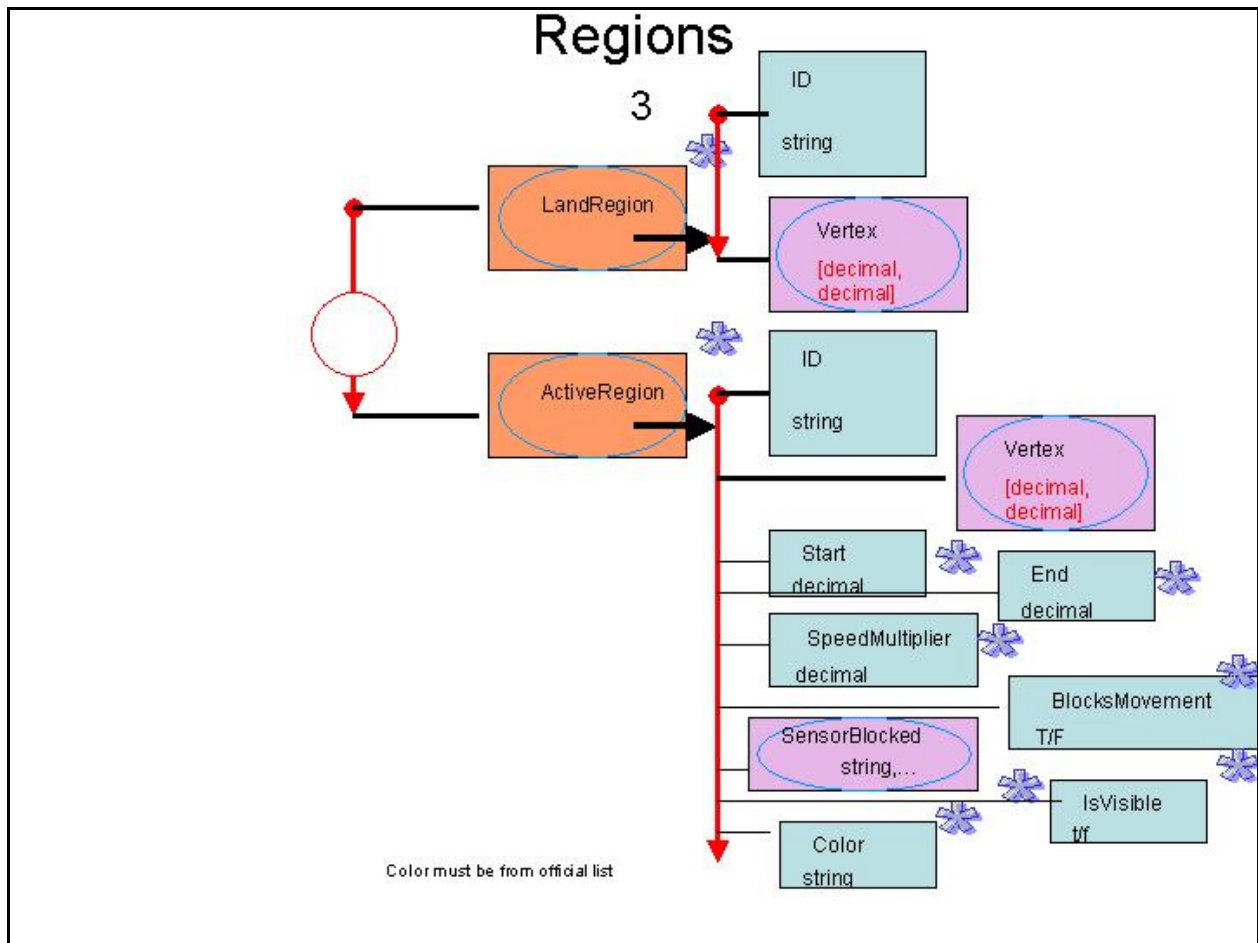
Tag	Appears on page	Within	Purpose
Name	Score		Name of score
Rules	Score		List of rules for score
Applies	Score		List of DMs that can be substituted as units or objects of a rule
Viewers	Score		List of DMs that will see this score displayed
Initial	Score		Initial value for score

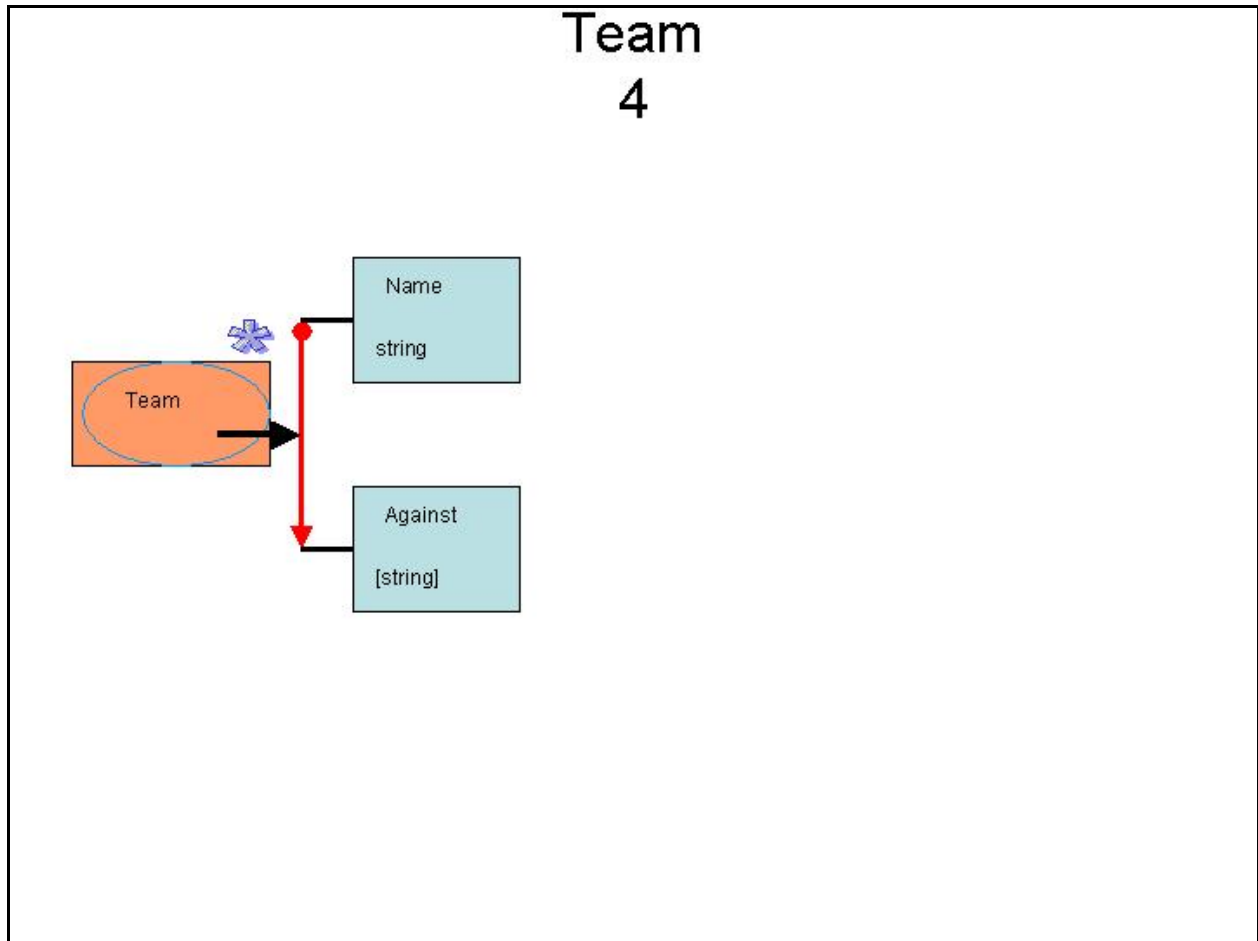
Diagrams

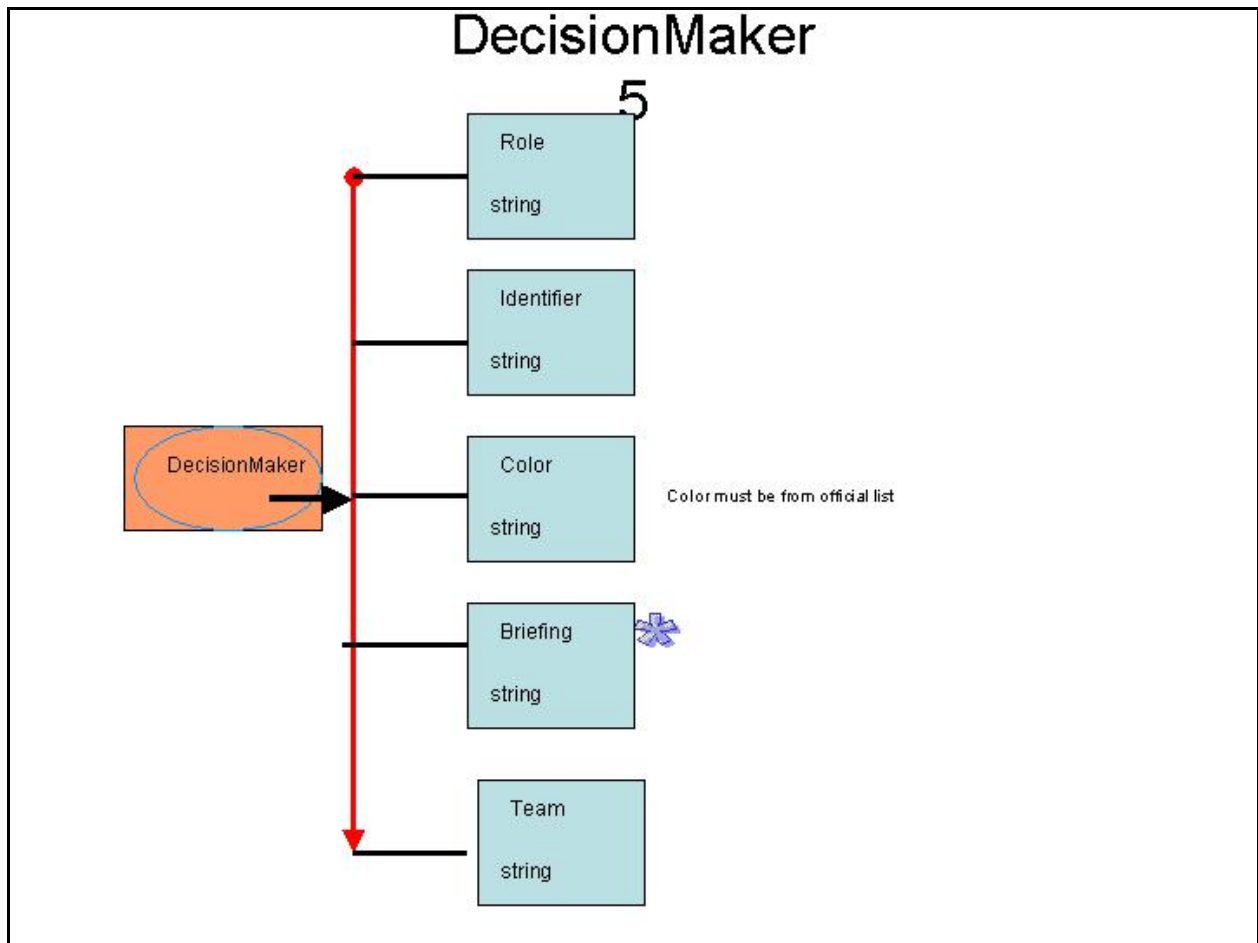


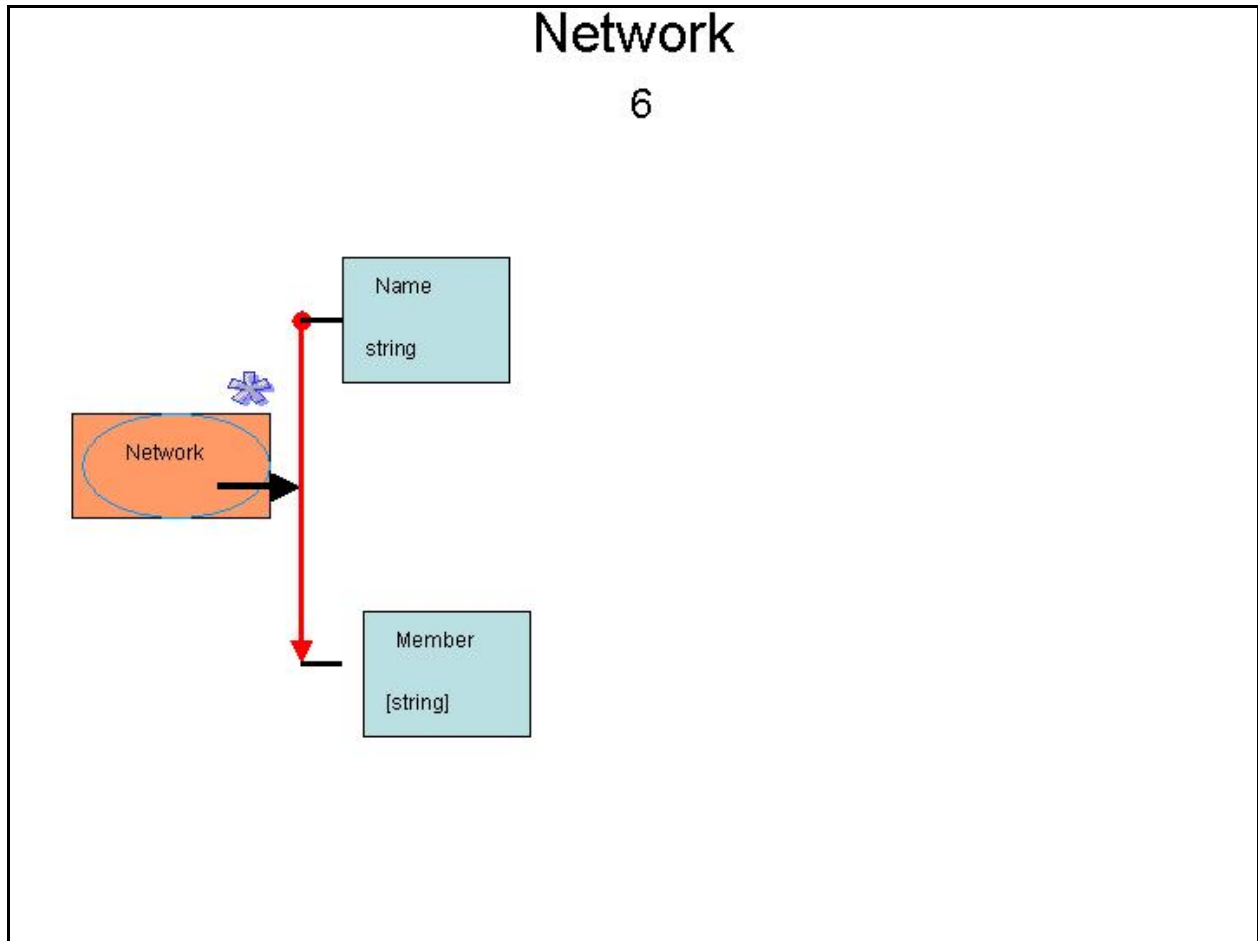


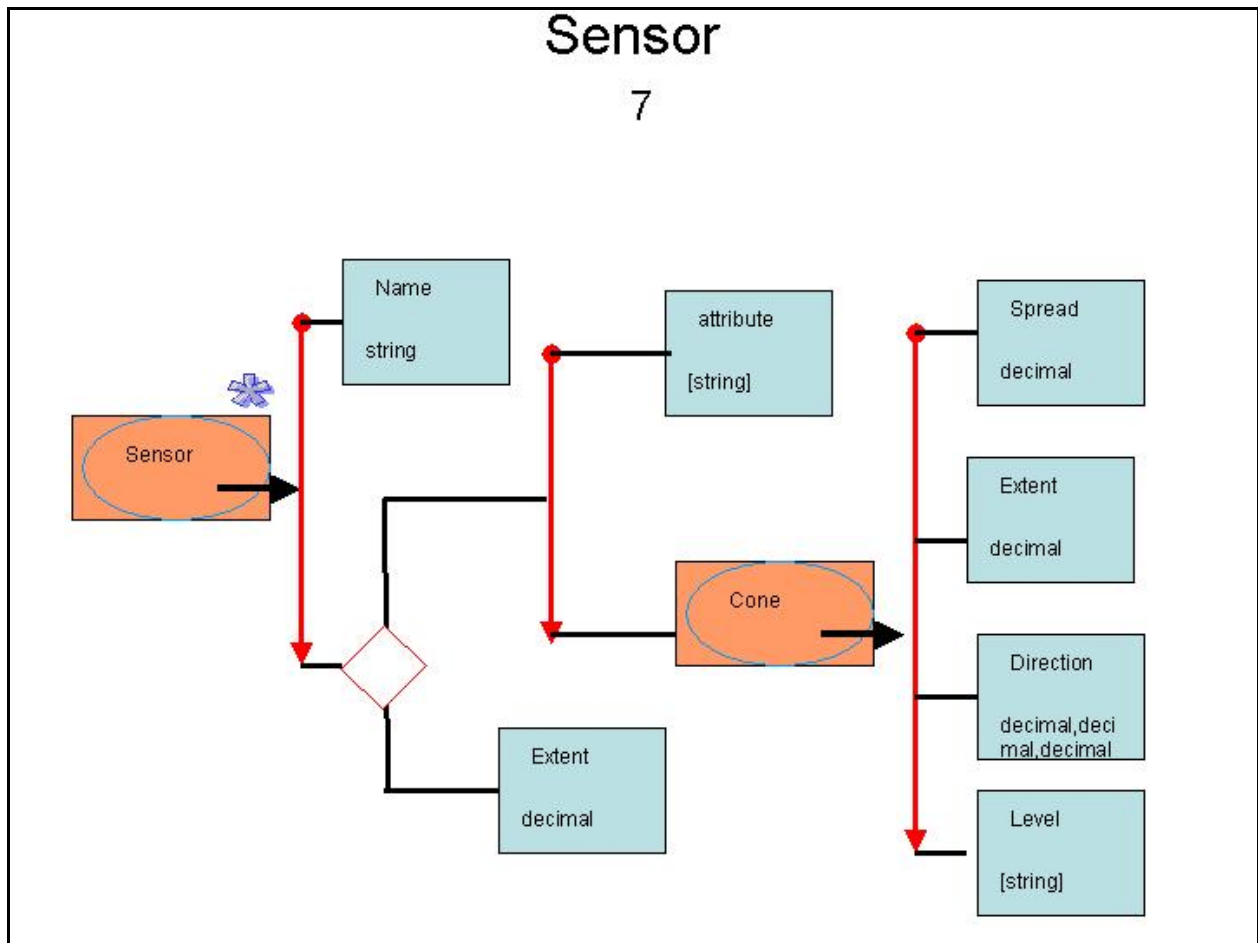


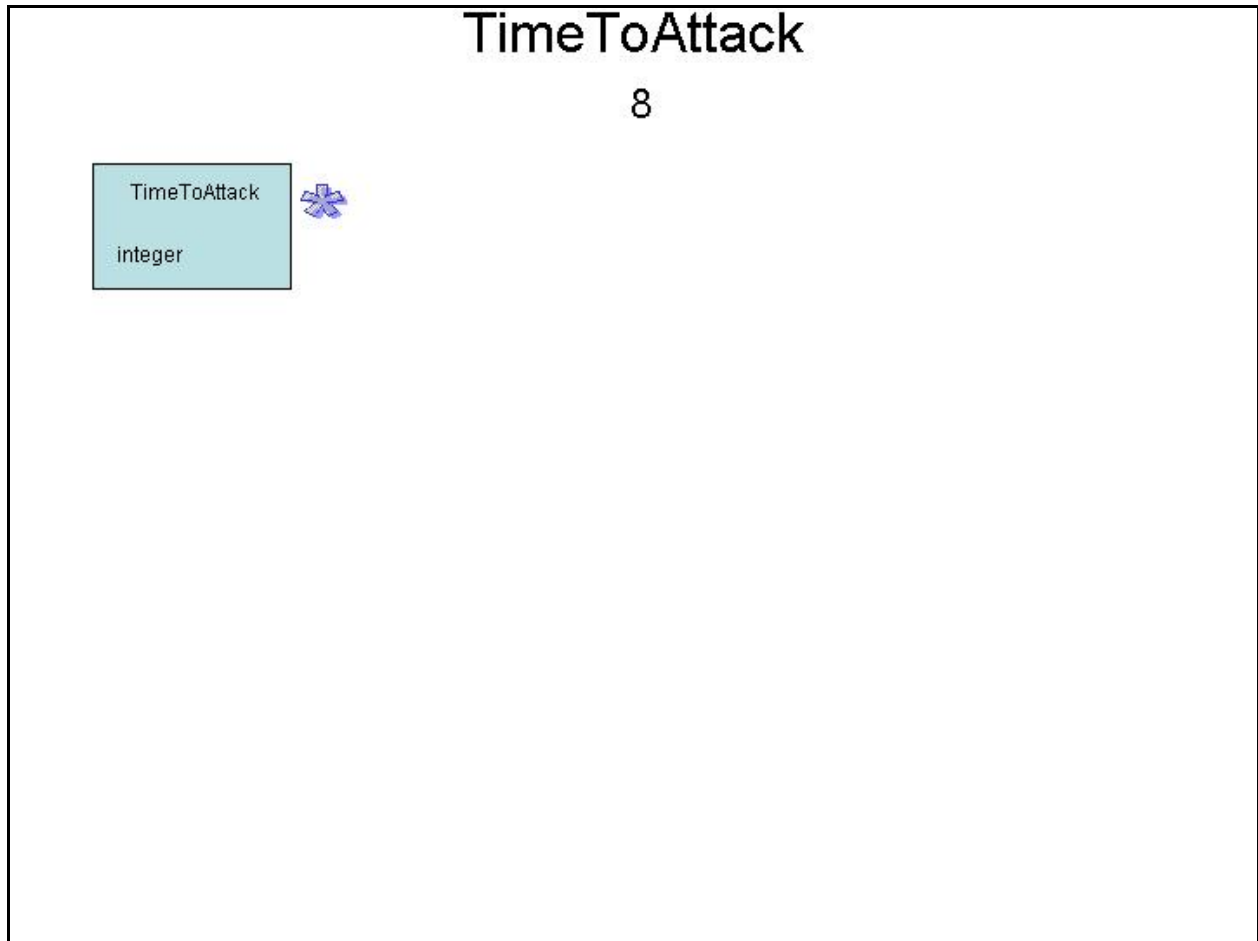


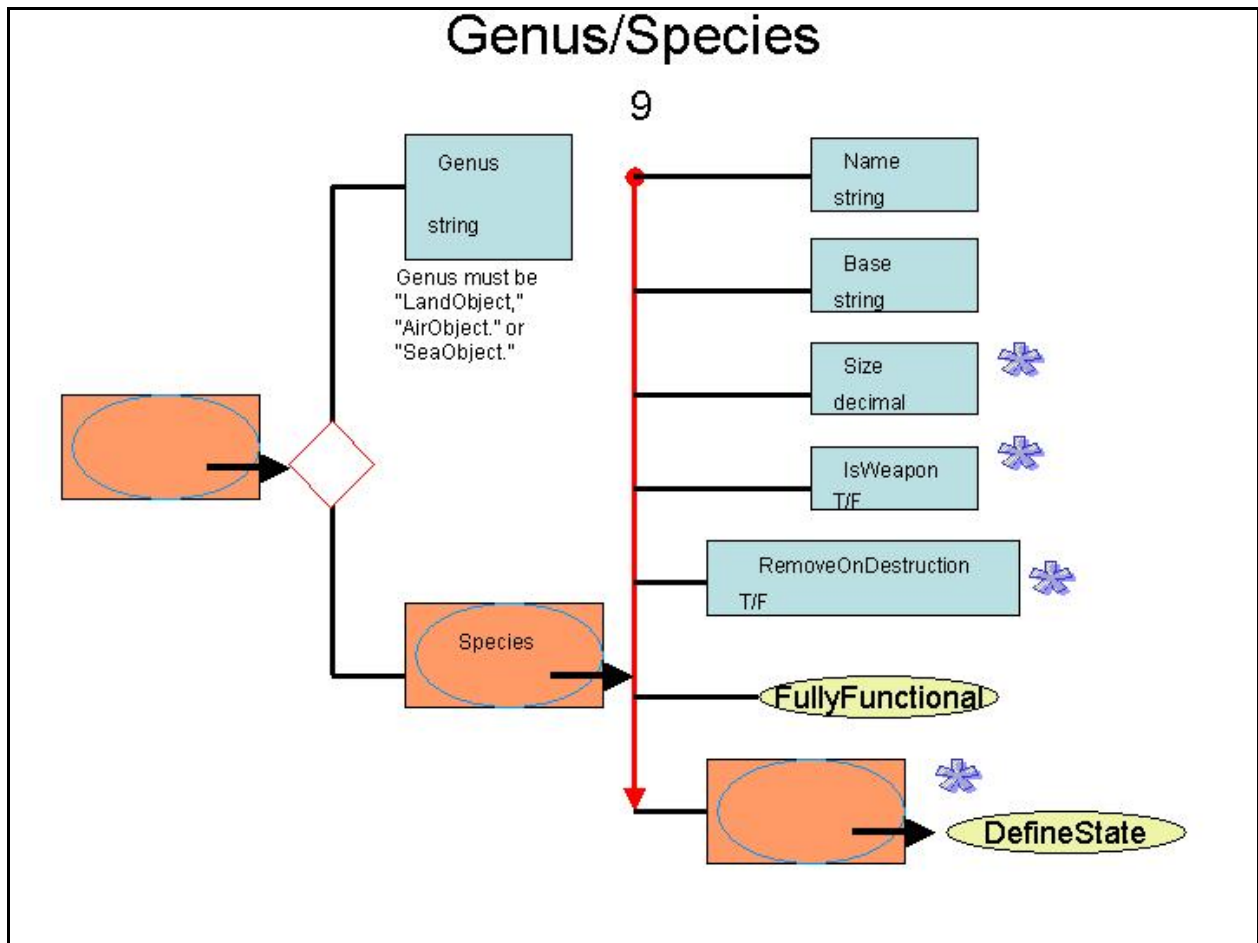


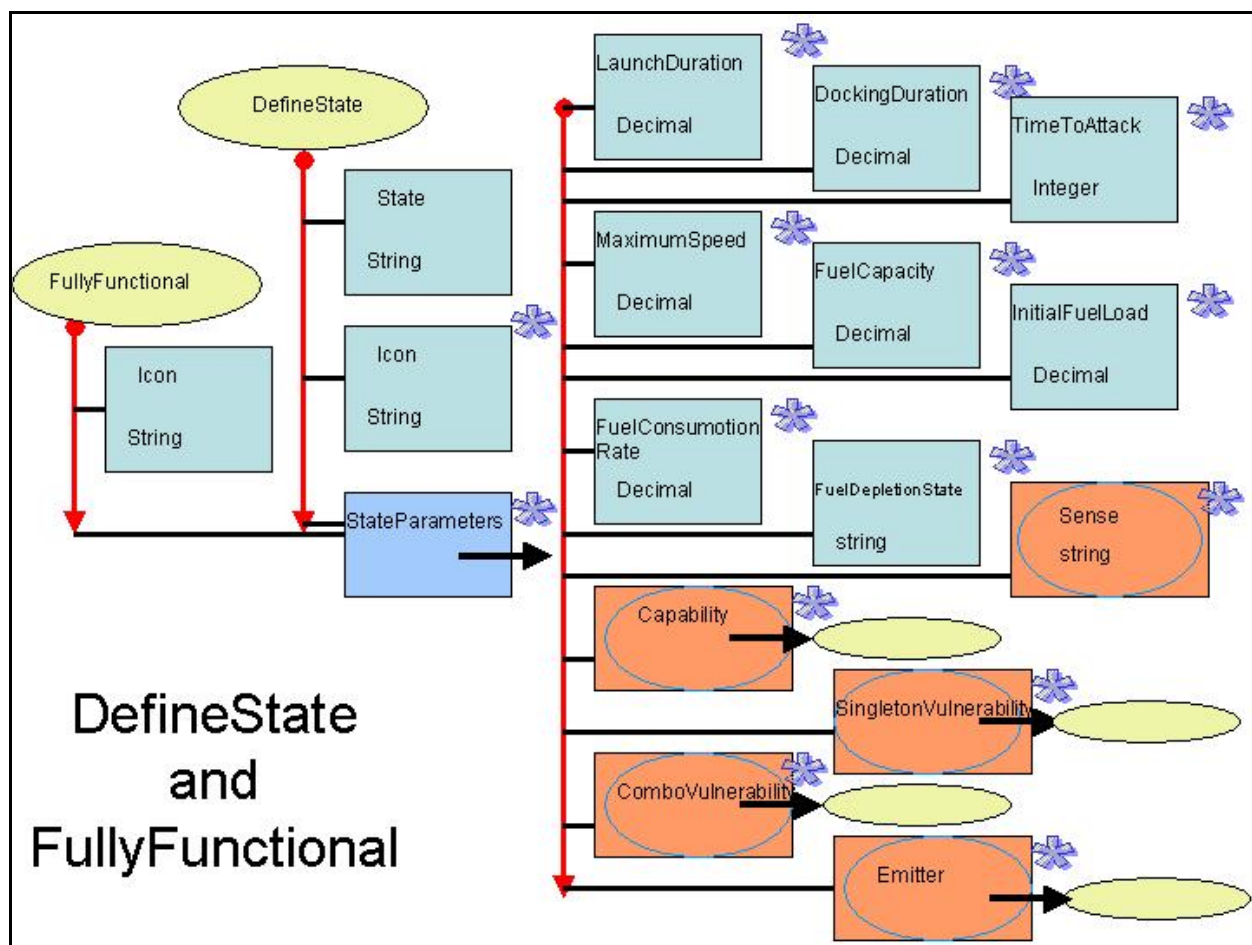


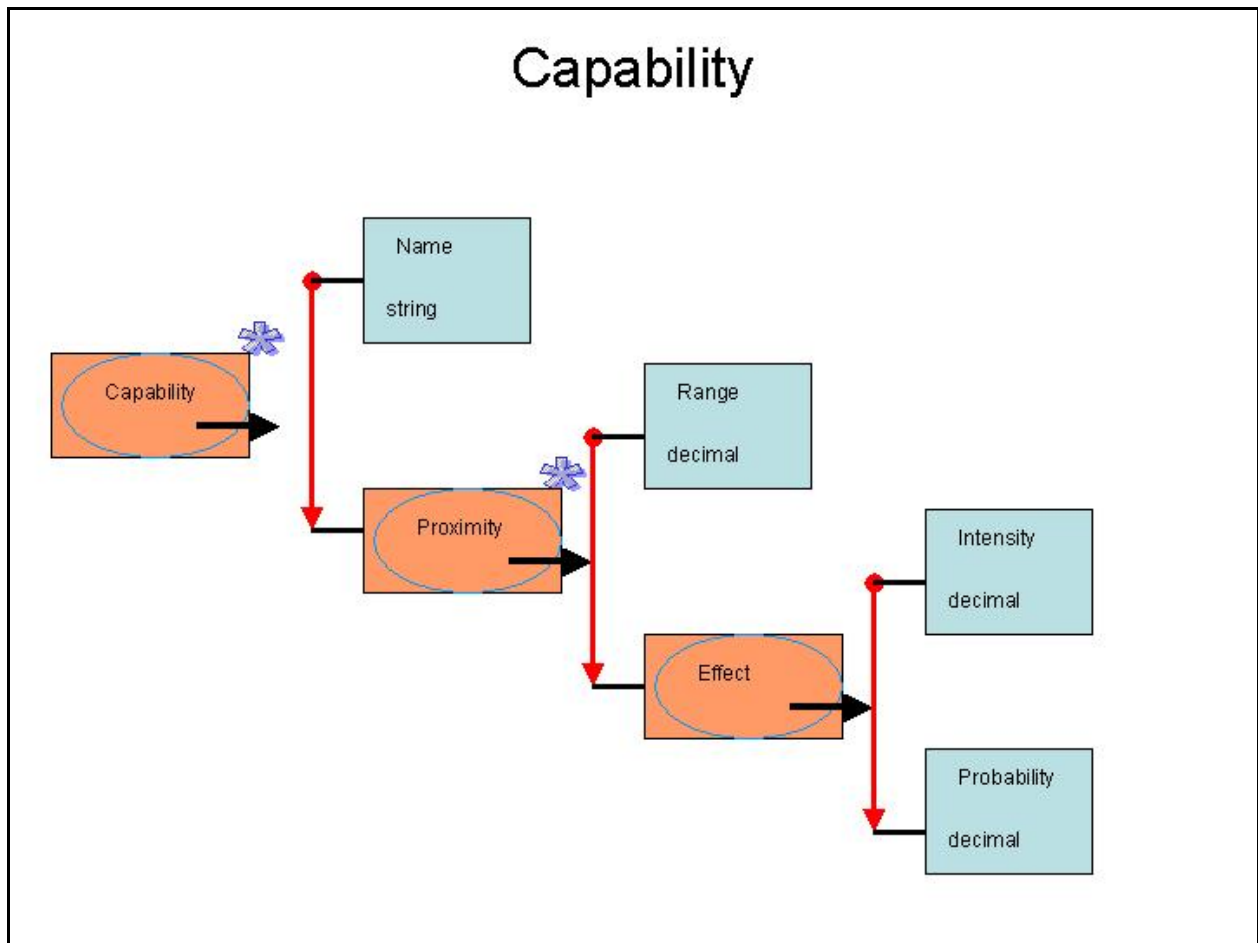


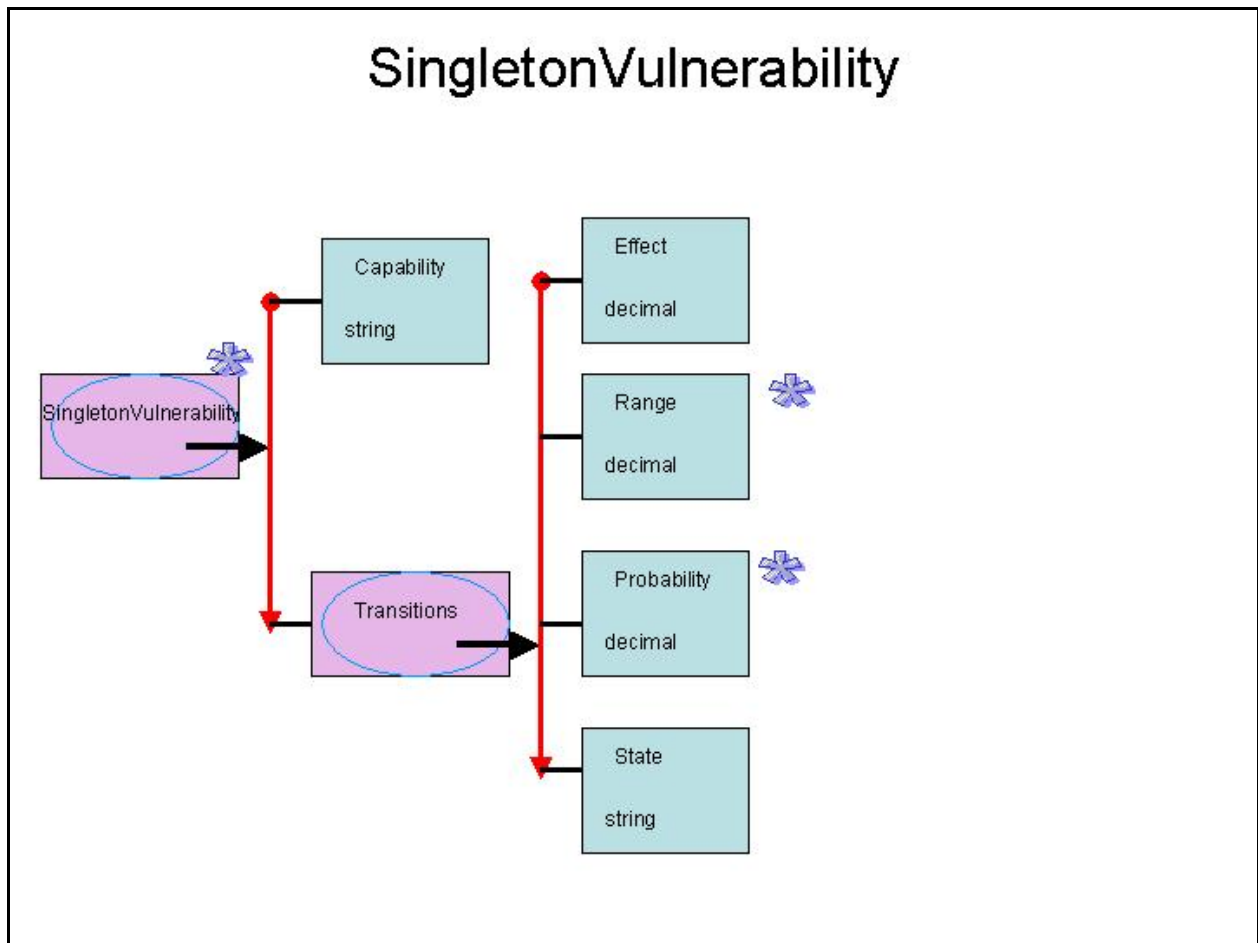


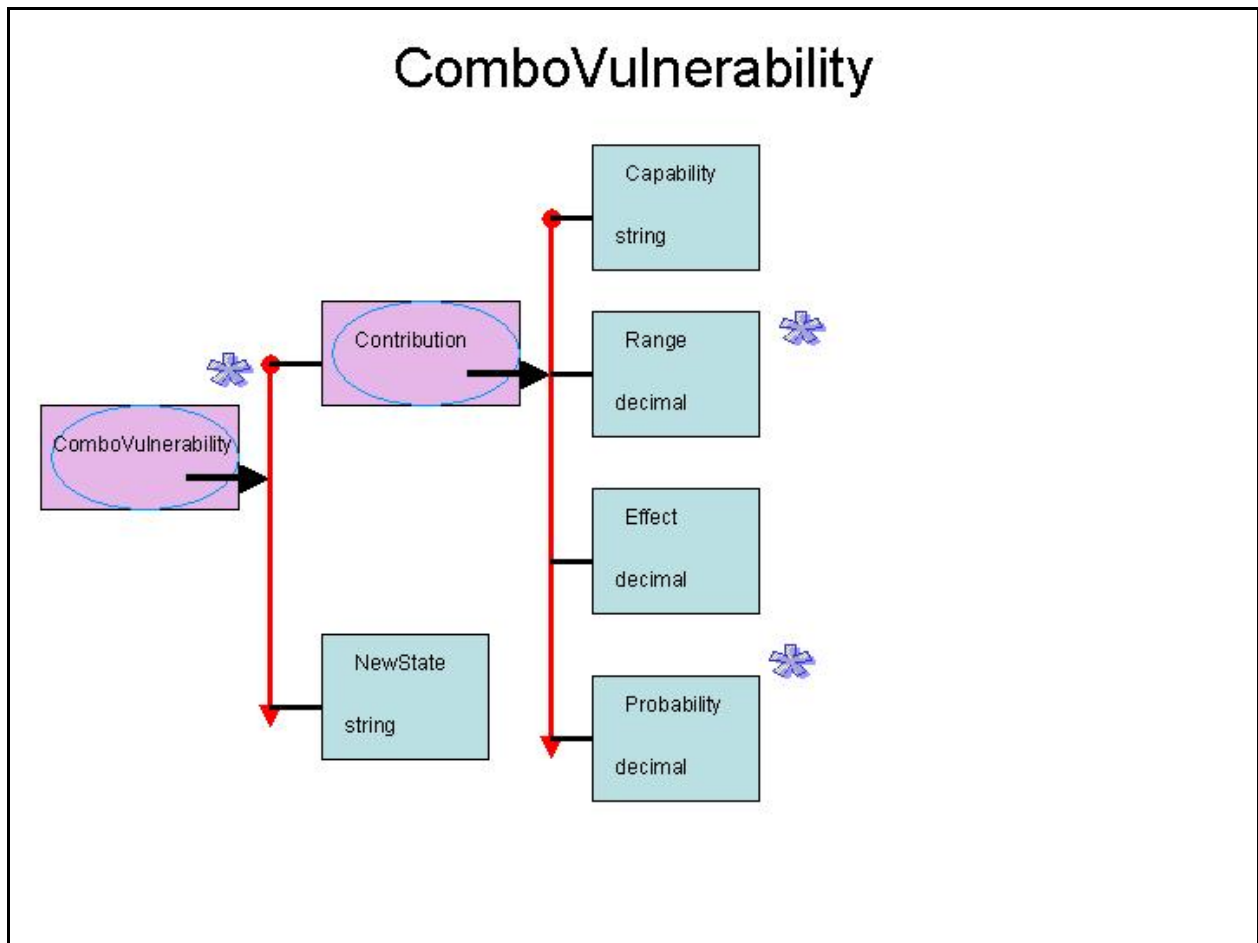


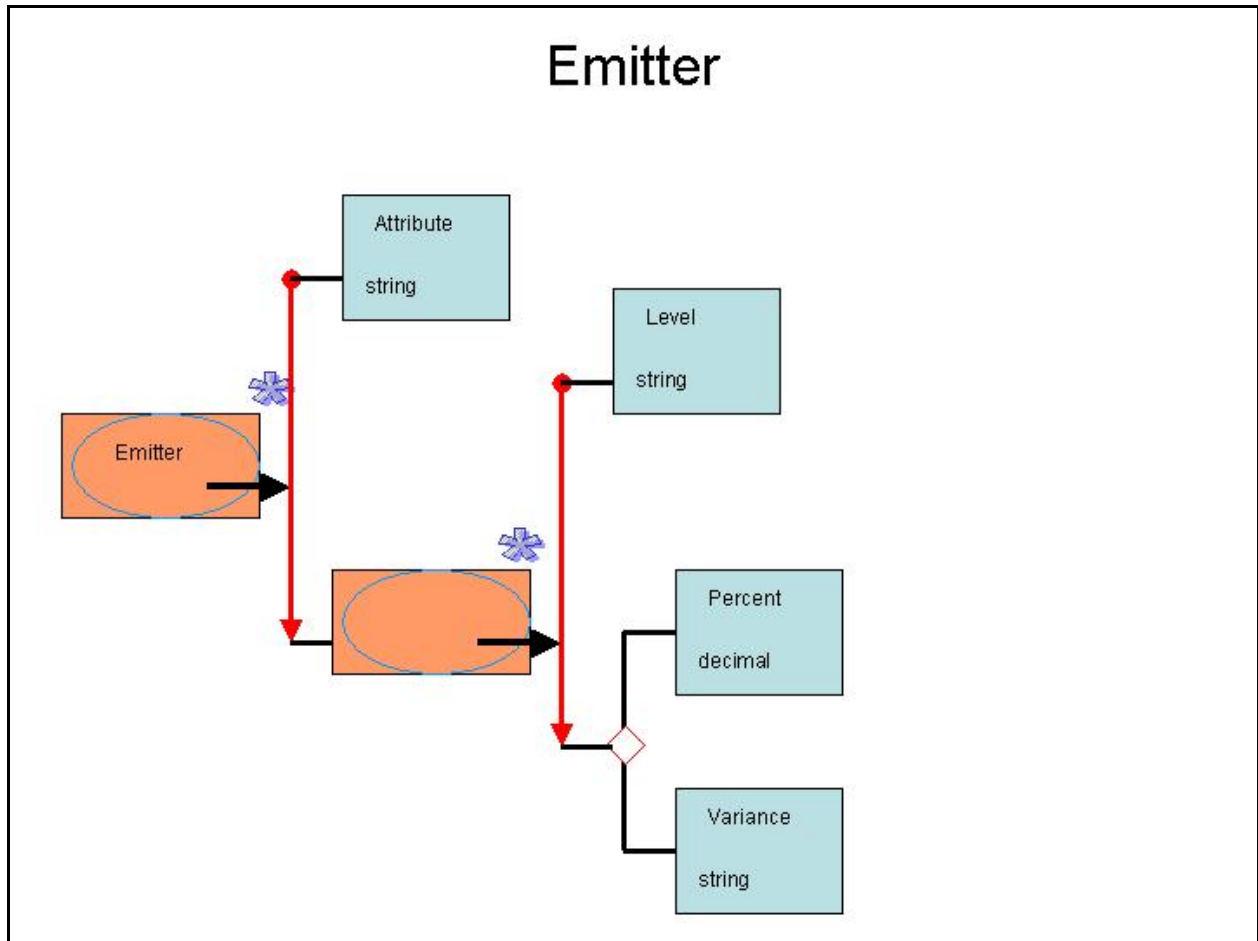


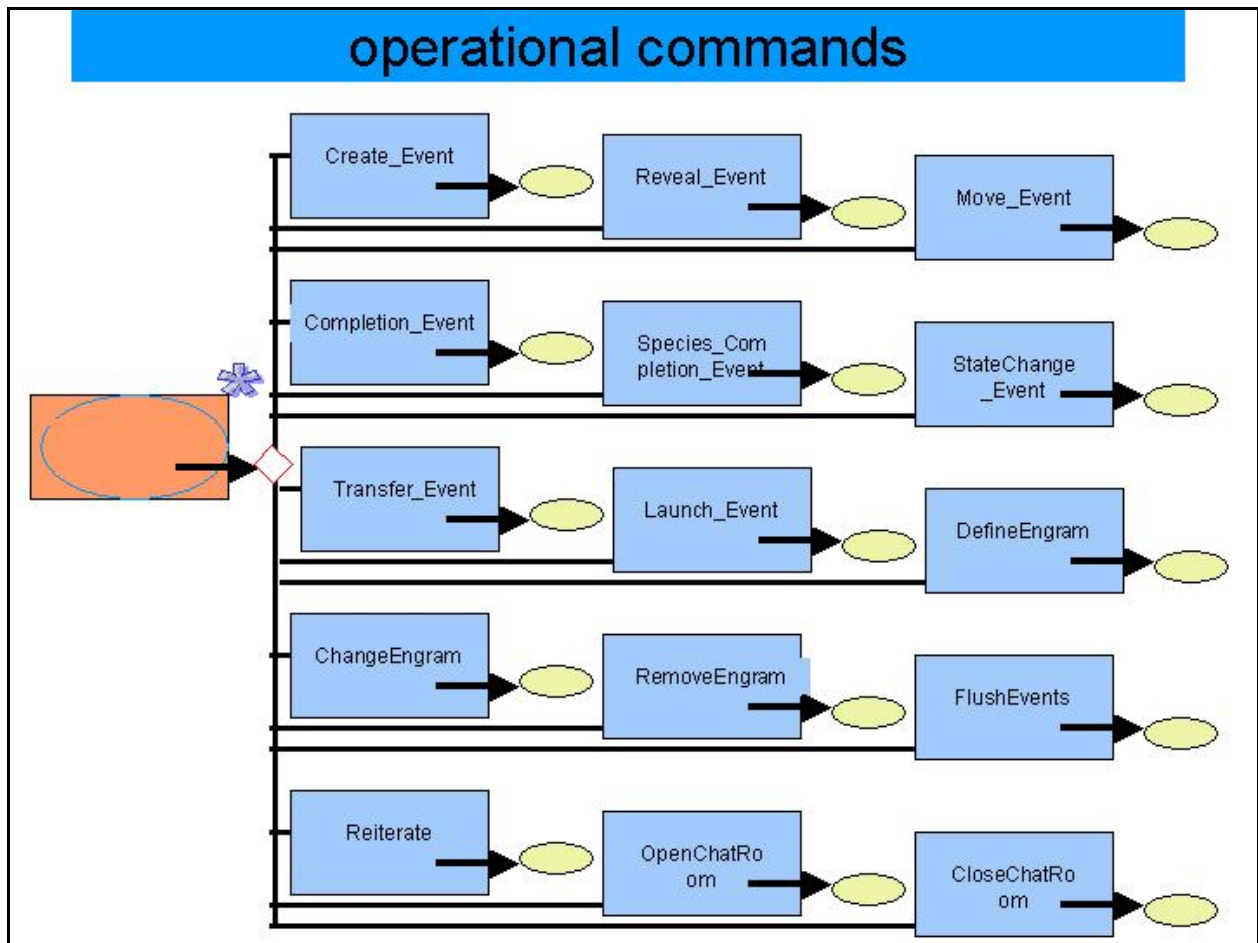


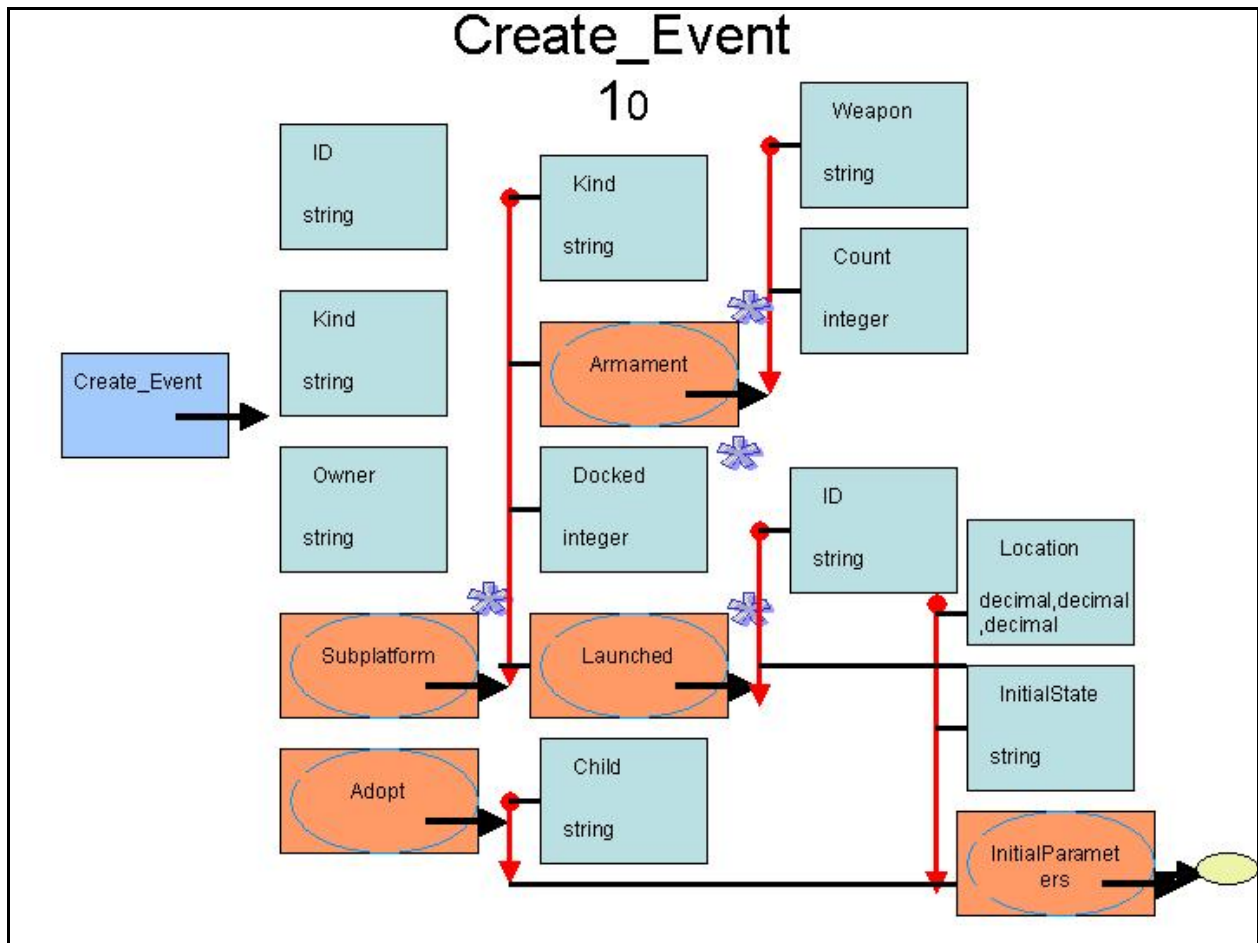












InitialParameters, StartupParameters

