AN2509 – Using the MACE API to Emulate Stochastic Effects: Pk

May 2025

Battlespace Simulations.
1229 Oak Valley Drive
Ann Arbor, Michigan 48108
Phone: 210-857-9672

If you have questions or comments, please contact us at support@bssim.com.

Overview

MACE is a physics based, high fidelity, multi-domain combat simulation environment, but there are times when a stochastic, effects-based outcome is desired. One area that our customers frequently ask for such an approach is in the air-to-air domain, specifically in the application of Pk or "probability of kill" tables. This application note will illustrate how to calculate a Pk table and apply it to a scenario by leveraging the MACE API event system.

An explanation of Pk

"Pk" or "P sub k" is the probability that a weapon will guide, fuse, and inflict lethal damage to an enemy aircraft. The Pk value can be influenced by many different factors ranging from the physical reliability of the missiles components to the robustness or complexity of the electronic warfare environment. In training scenarios, or when analizing tactics, different Pk values will be applied to reinforce learning objectives or to test the robustness of tactics. It is a stochastic value, in that it is well described by a random probability distribution; in practical terms, this means it is a statistical value that can be evaluated at the appropriate time in the scenario to determine the success or failure of the event – in this case, the success or failure of a missile fired against an enemy target.

In practice, there are generally two ways in which Pk values are applied. In the first, a "dice roll" is applied at missile timeout and Pk enforced based on the outcome at that time. This is the most accurate application of Pk.

In instances where shots need to be correlated among live, virtual, and constructive participants and Pk values applied through an expected distribution of shots, pre-calculated "shot tables" are frequently used. This is not the most realistic application of shot probabilities and can artifically alter outcomes by enforcing Pk distributions. For example, say a fighter takes two shots, each with a Pk of 50%. It is entirely possible that both shots will function properly and result in a kill, or for neither shot to result in a kill; the "coin flip" that decides the outcome is applied to each missile individually. The law of large numbers dictates that we would approach our 50% distribution as the number of shots increases, and the more shots we consider the closer to 50% we can expect to converge (i.e. the "sample mean" would converge to the "true mean"). When we pre-compute a table of shots and apply enforce a 50% distribution, we never test the worst case edge case where no shots would function, or even when all of the first volley fail. However, shot tables do offer advantages in allowing coordination of shots during a training exercise.

We will consider both approaches in this application note.

Applying Pk in MACE

MACE being a physics based simulation, the actual flyout and guidance of the missile is determined frame by frame in MACE; the weapon will detonate if and when its detonation proximity threshold is reached against a target, and damage will be applied to the enemy target(s) in accordance with the weapon damage properties as defined in its weapon configuration file. We can, however, apply a "dice roll" when the weapon damage is calculated and, comparing the outcome of our dice roll to a pre-calculated distribution of hits and misses, calculate whether or not the weapon would actually inflict any damage on its target.

Applying Pk using the `WeaponDamage` event

There are a number of weapon related events plugins, codescripts, or commands can subscribe to. One of those is `IMission.WeaponDamage`, which is raised when MACE has calculated a damage percentage for a weapon, but before that damage has been applied. By listnening for that event, we can cancel the application of damage altogether or substitute a "custom" damage percentage for MACE to apply. We can subscribe to the WeaponDamage event as follows:

```
MissionInstance.Mission.WeaponDamage += HandleWeaponDamage;
```

The `HandleWeaponDamage` event itself would use a pseudo-random number generate (declared within the namespace) to calculate a double precision floating point value between 0.0 and 1.0 and compare it against a desired Pk value (declared within the namespace):

```
private Random _rnd = new Random();
private const double _pk = 0.5;

public static void HandleWeaponDamage(object sender, EventArgs e)
{
    try
    {
        IMission.WeaponDamageEventArgs args = e as
IMission.WeaponDamageEventArgs;
        if (args != null)
        {
            if (args.MunitionEntity.Type.StartsWith("AIM-120"))
            {
                if (_rnd.NextDouble() > _pk)
                {
                    args.Cancel = true;
                }
            }
        }
    }
    catch (Exception ex)
    {
        MissionInstance.Mission.Logger.ErrorMessage(ex);
    }
}
```

Calculating a "shot" table

We can revise our approach to accommodate a pre-calculated "shot" table. The size of the table can be tailored to a reasonable size that's in line with the number of missiles we expect to use in a given scenario. For example, if we're computing a lookup table of pre-computed Pk values for AMRAAM shots, and our scenario has 4 aircraft each carrying 4 AMRAAM missiles, a reasonable size for our lookup table would be 16 possible shots. Using a Pk for the AMRAAM of 50% mission wide, we would expect 8 of our shots to hit their target and inflict lethal damage, and 8 to "miss". Translating that to code, we might have something like the following

```
public const int n = 16;
public static bool[] _hitTable = new bool[n];
```

```
public static double pk = 0.5;
public static int truesRemaining = (int)Math.Round(pk * n);
public static Random rnd = new Random();
public static int slotsRemaining = n;
public static int detonationCount = 0;

for (int i = 0; i < n; i++)
{
    // pre-generate a hit table of hits and misses
    double pHit = (double)truesRemaining / slotsRemaining;
    if (rnd.NextDouble() < pHit)
    {
        _hitTable[i] = true;
        truesRemaining--;
    }
    else
    {
        _hitTable[i] = false;
    }
    slotsRemaining--;
}
```

The calculation of the table could be made more dynamic by iterating through all entities in the mission and counting up the number of missiles in the scenario, and sizing the table and its entries dynamically, but for illustration purposes the code as written will give us a pre-calculated table of 16 possible “hits” or “misses”, pseudo-randomly distributed in our table.

In the modified `HandleWeaponDamage` event handler, we first check to see if the weapon that’s detonating is the weapon type we care about. In this example, we’re applying it only to any AIM-120 AMRAAMs. Once we validate that the weapon event is one we should apply a Pk value to, we keep track of the number of weapon detonations that have occured and look up the appropriate entry in our pre-calculated lookup table to determine if the event should be counted as a “hit” or a “miss”. If our table says it’s a “miss”, we simply cancel the event. MACE will not proceed with the actual application of the damage calculation once the event is cancelled, and we will have successfully achieved the desired effect of a “miss” based on our designated Pk.

```
public static void HandleWeaponDamage(object sender, EventArgs e)
{
    try
    {
        IMission.WeaponDamageEventArgs args = e as
IMission.WeaponDamageEventArgs;
        if (args != null)
        {
            if (args.MunitionEntity.Type.StartsWith("AIM-120"))
            {
                if (!_hitTable[detonationCount])
                {
                    // cancel the application of damage
                    args.Cancel = true;
                    MissionInstance.Mission.LogMissionEvent($"Pk miss
calculated against {args.TargetEntity.Name}.");
```

```
                }
                else
                {
                    MissionInstance.Mission.LogMissionEvent($"Pk hit
calculated against {args.TargetEntity.Name}.");
                }
                detonationCount += 1;
            }
        }
    }
    catch (Exception ex)
    {
        MissionInstance.Mission.Logger.ErrorMessage(ex);
    }
}
```

This approach can be applied via a codescript, in a custom plugin, or even in a weapon command that in turn exposes a weapon property for setting the Pk.