14.2 Background and Terminology

Introduction

The following terminology is used throughout this document.

- **System**: A radio system of SEAMCAT. Currently SEAMCAT supports two kinds of systems, i.e. Generic System and Cellular System (with four sub types available: CDMA UpLink, CDMA DownLink, OFDMA UpLink, OFDMA DownLink). However different, all systems has common traits such as frequency, receiver characteristics, transmitter characteristics, positioning, etc.
- **Victim system**: referred to as 'The victim' as there is only one victim for a simulation.
- **Interfering system**: a system that interferes with the victim system.
- **Interference link**: this is a constellation of a victim system and an interfering system. But it also includes configuration outside of systems, i.e. relative positioning of the systems, propagation model, specifics for system types such as user defined DRSS for generic system.
- **Scenario**: a complete configuration of a victim system and a list of interference links (and thereby a list of interfering systems).
- **Event** (or snapshot): a random trial of all distributions in a scenario.
- **EventResult**: from an event the system simulations apply calculations and produce a number of results, which is what we call the event result.
- **Simulation result**: the total result of combining all produced EventResults.

14.2.1 Understanding abstraction level

We can conceptually view SEAMCAT as a layered cake of abstractions. Knowing these different layers helps a lot in understanding how to navigate the source and how to understand SEAMCAT.

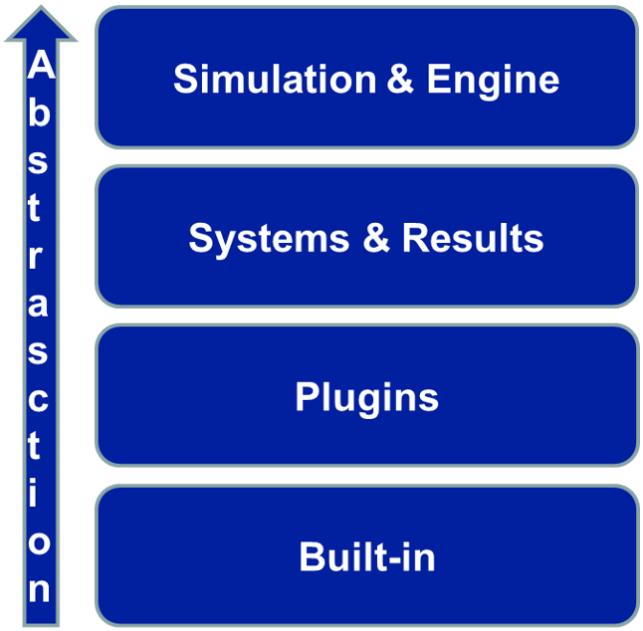

Figure 299: Abstraction level in SEAMCAT

So in a very high level of abstraction, the Monte Carlo simulation runs through the specified number of randomised events, which each produces an EventResult. Lastly it combines all these EventResults into a simulation result which can be inspected in SEAMCAT.

14.2.2 Simulation & Engine

We start at the highest abstraction: Simulation & Engine. If we forget about the specific system types and multiple interferers we get the conceptualized view of a scenario. A victim system can be interfered by an interfering system.

On a very high level of abstraction an interference simulation consists of

1. victim.simulate, i.e. Configure the victim system;
2. interferer.simulate, i.e. Configure the interference system;
3. and collect the interference results based on the settings of the two systems.

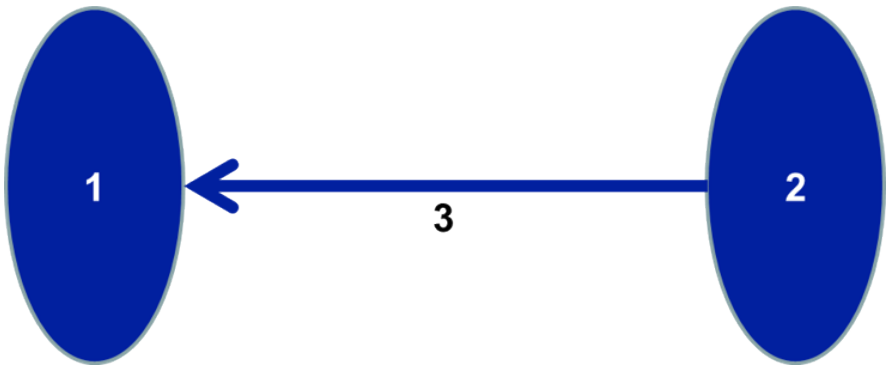

Figure 300: Abstraction level of an interference simulation

A closer look at the 3 steps: get the list of active receivers from the victim. Remember do not think of a specific system, if you for instance think of a Generic system you might say that this is never the case because there can be only one receiver. But there being only one is a special case of being many. And for Cellular systems there can definitely be many receivers (UE or BS). So the highlevel view of the algorithm is that there is a list of active receivers in the victim system.

Now we do the same (but opposite) for the interfering system: get the list of active transmitters. Again the list of active transmitters is the most general or highlevel notion of the algorithm which describes all possible cases (0, 1, or many).

Finally we combine all victim receivers with interfering transmitters and base the calculation of the interference with this as shown in Figure 301.

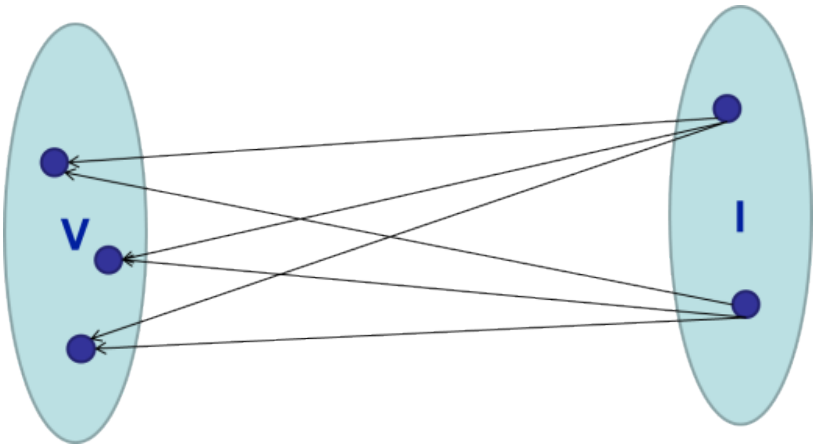

Figure 301: Abstraction level of IT->VR links

These IT->VR links are really what we are looking for and bases the interference calculations upon.

Given this the following results are possible:

- 0 – For some reason there was either 0 active receivers or 0 active transmitters in the victim, interfering system (respectively). Therefore we have no IT->VR links;
- 1 – There is precisely one active receiver and one active transmitter. This is most commonly the case for ”generic-generic” simulations;
- Many – There are many IT->VR links either because the systems are generic with many active transmitters or cellular (or maybe it is a future system not invented yet).

We have certainly described a lot about how SEAMCAT works but at no point it was needed to talk about any systems specifics. Of course during a simulation the choise of system will very much affect how it is done, but conceptualized like this there is no need.

On the highest level of abstraction we have the Simulation & Engine. “Simulation” holds the scenario setup and various hook for starting and completing a snapshot while “Engine” runs a simulation by controlling the steps in the simulation

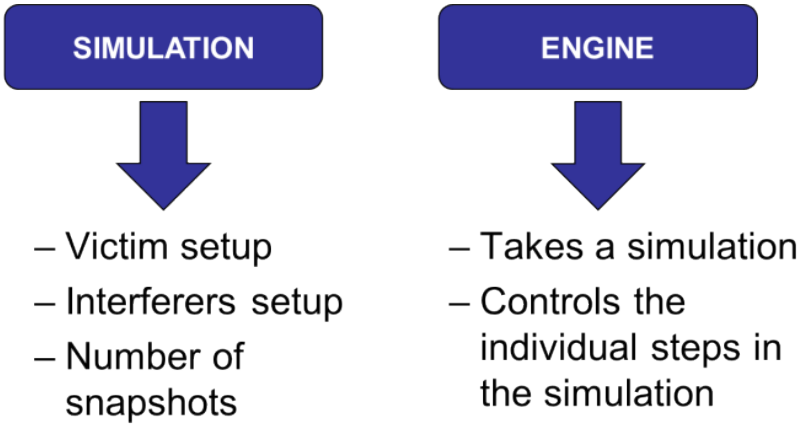

Figure 302: Simulation and Engine abstract representation

The figure below from the engine shows almost exactly the steps described above.

The real implementation is in InterferenceSimulationEngine.java take a look at it. Remind yourself that programming is all about details: there are so many other things going on than the actual loop.

```
public MutableEventResult single( VictimSystemSimulation victimSimulation, int eventNumber, Simulation simulation ) {
    fixSeed( simulation, eventNumber);

    MutableEventResult eventResult = new MutableEventResult(eventNumber, (WorkspaceScenario) simulation.getScenario());
    victimSimulation.simulate(eventResult);

    List<InterferenceLinkSimulation> iSims = new ArrayList<>();
    for (InterferenceLink interferenceLink : simulation.getScenario().getInterferenceLinks()) {
        InterferenceLinkSimulation iSim = simulation.getInterferenceLinkSimulation(interferenceLink);
        iSims.add(iSim);
        Point2D victimPos = victimSimulation.getSystemPosition( eventResult, interferenceLink);
        iSim.simulate(simulation.getScenario(), eventResult, interferenceLink, victimPos);
    }

    victimSimulation.collect(eventResult);

    simulation.eventComplete(eventResult, victimSimulation, iSims);

    return eventResult;
}
```

Figure 303: Engine code

The engine asks the simulation for the victim system simulation. So the simulation controls which victim simulation instance is handed to the engine, and the engine then uses that instance for doing its calculation. Therefore the engine is decoupled from the knowledge of which specific victim simulation is running (e.g. Cellular victim).

14.2.3 Packages

SEAMCAT has different modules. These are used to group certain classes of the code and handle dependencies between them. This is similar to the layers of abstraction where the engine only knows general system types but not specifics. Modules can be used to enforce such dependencies.

There are two main modules: model and application.

- Model holds all the abstract definitions that composes a System – Transmitter, Receiver, Antenna, Propagation model, Distribution, etc.;
- Application implements the SEAMCAT UI, simulations, Propagation Models, and basically ties everything together. We will descuss this structure in more detail later.

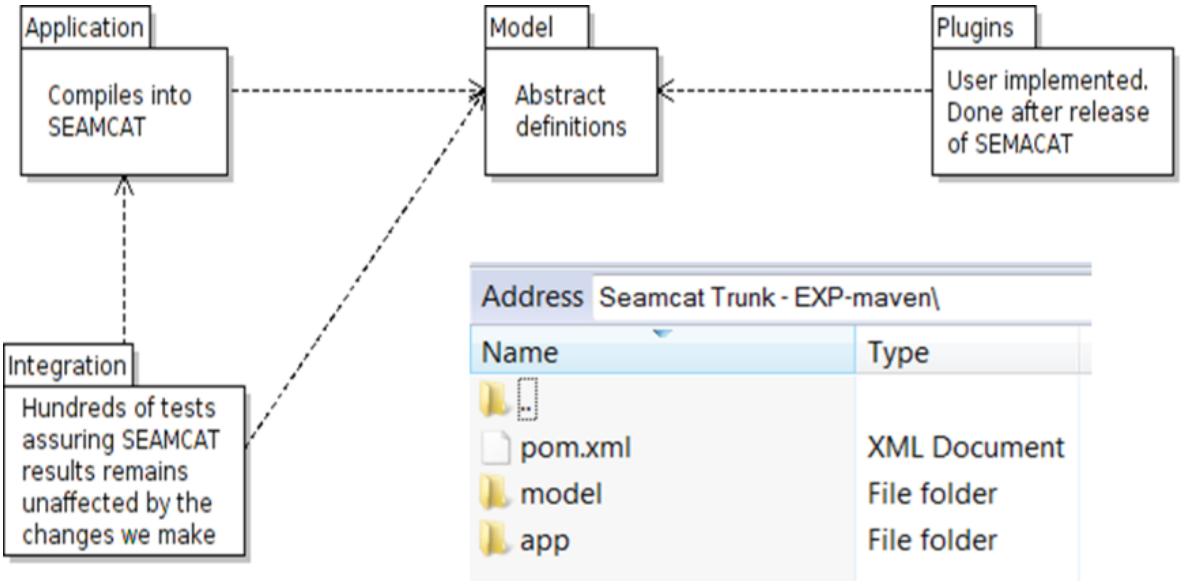

Figure 304: Modules of SEAMCAT

Inside a module we organise the software files in packages. It is similar to a folder and like a folder it contains the individual source code files. The names of the packages is helpful when nagivating the source code.

The packages of the model module:

- The names can be helpful when navigating the source code;
- Cellular and generic contains the definitions for the cellular and generic systems;
- Plugin holds the plugin definition for the mentioned types (Eventprocessing, Coverageradius, propagation etc…);
- Distribution, functions, and mathematics holds the distributions and other helper functions;

- Simulation and types holds the simulation results and general type definitions (Antenna, Receiver, RadioSystem, etc).

The packages of Application is more messy than the model. So here it is harder to know exactly where to go. The packages to know here is: scenario and simulation:

- Scenario contains the implementation of some of the types defined in Model;
- Similarly with simulation: here are the specific simulations e.g. CDMADownlinkVictimSimulation.

It is the very nature of code to change, but with a good design and understanding of the abstractions it should be easy to identify part that will change the least. Most of the dynamic parts of the code can be examined with runtime inspection by software development tools. Mechanism vs Policy is a software principle based on the fact that mechanism change infrequently whereas policies change all the time.

Mechanism is here the engine of SEAMCAT and policies are the plugins. The parts that changes behaviour are left open for developers.

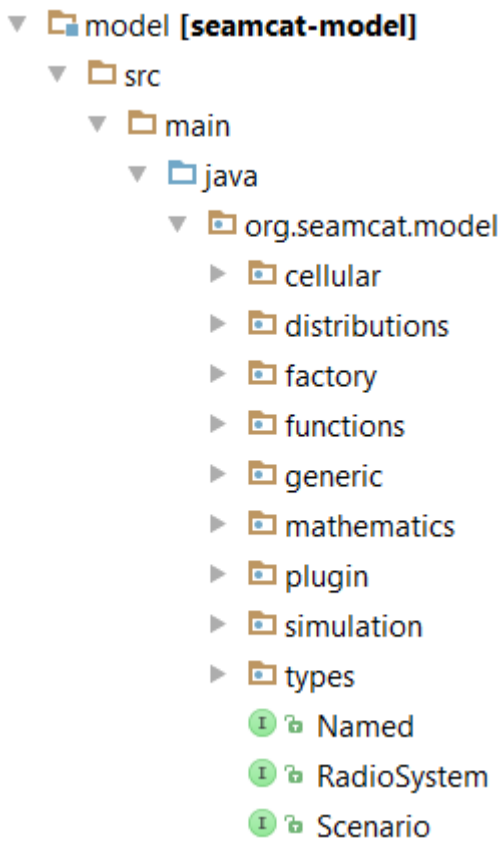

Figure 305: Packages of Model

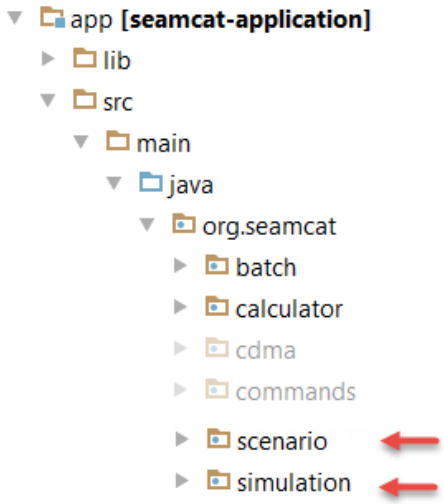

Figure 306: Packages of Application