

Complex Simulation Experiments Made Easy

Advanced Tutorial

TOM WARNKE AND ADELINDE M. UHRMACHER

Modeling and Simulation Group
Institute of Computer Science
University of Rostock

This tutorial introduces the **Simulation Experiment Specification on a Scala Layer (SESSL)**¹, a unified tool for simulation experiments

- with any simulation model and
- with any experimental setup
- that can be easily replicated.

SESSL's source code is freely available under the Apache 2.0 License at <http://sessl.org>.

¹R. Ewald and A. M. Uhrmacher. "SESSL: A Domain-specific Language for Simulation Experiments". In: *ACM TOMACS* 24.2 (2014). DOI: [10.1145/2567895](https://doi.org/10.1145/2567895).

Simulation
System

Experimental
Method

Simulation
System

Experimental
Method

Simulation
System

Experimental
Method

NetLogo

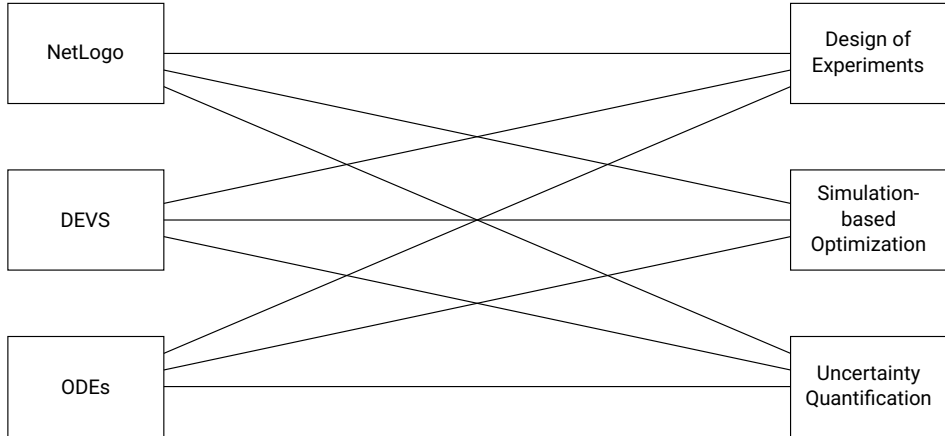
Design of
Experiments

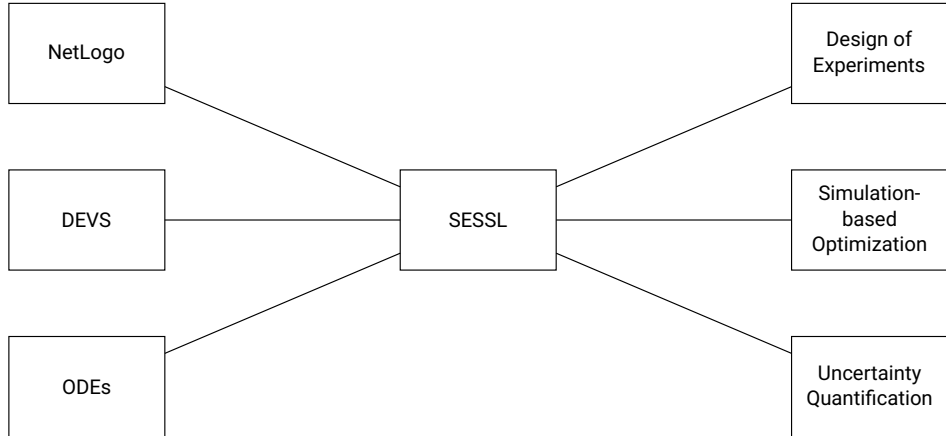
DEVS

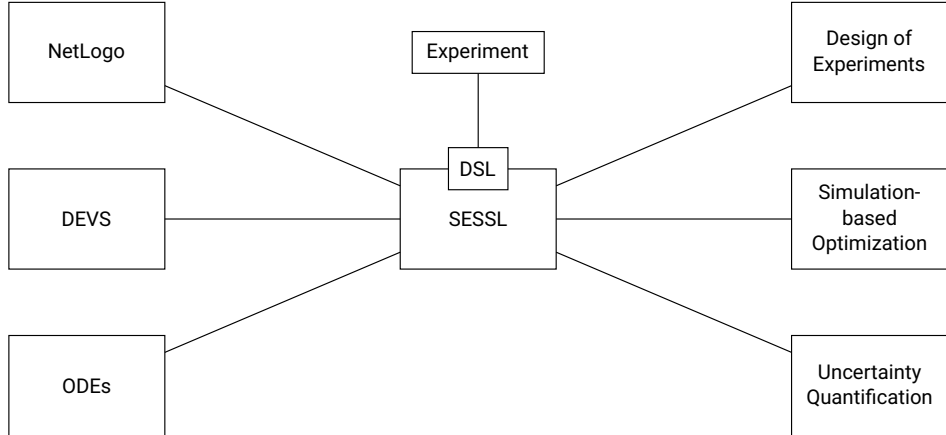
Simulation-
based
Optimization

ODEs

Uncertainty
Quantification

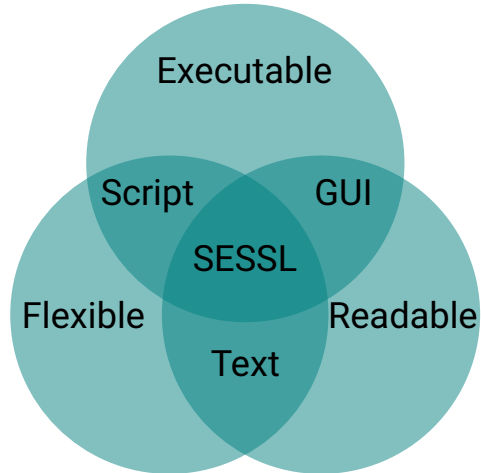








- If you develop simulation software, using SESSL gives your users access to many experimental methods.
- If you develop experimental methods, using SESSL makes your methods available for many simulation applications.
- If you are a modeler, using SESSL
 - gives you an **expressive domain-specific language (DSL)** for running various kinds of simulation experiments and
 - enables you to distribute executable experiments with your model, allowing for **replication of your results**.



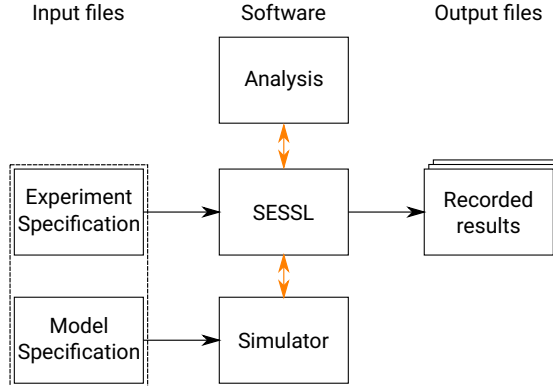
Key Design Concepts

Playing around with a simple simulation experiment

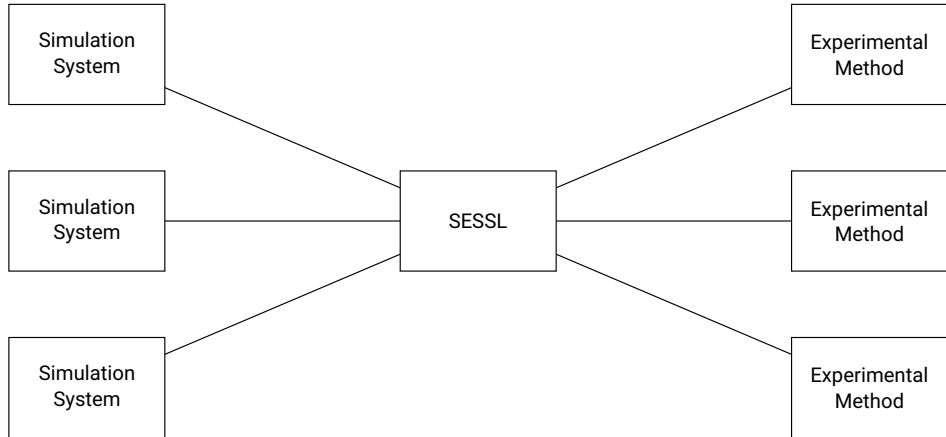
Complex simulation experiments

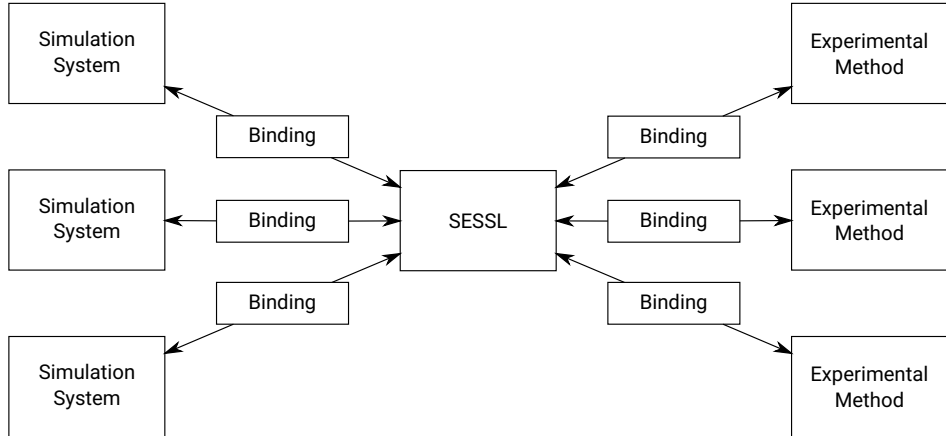
Extending SESSL

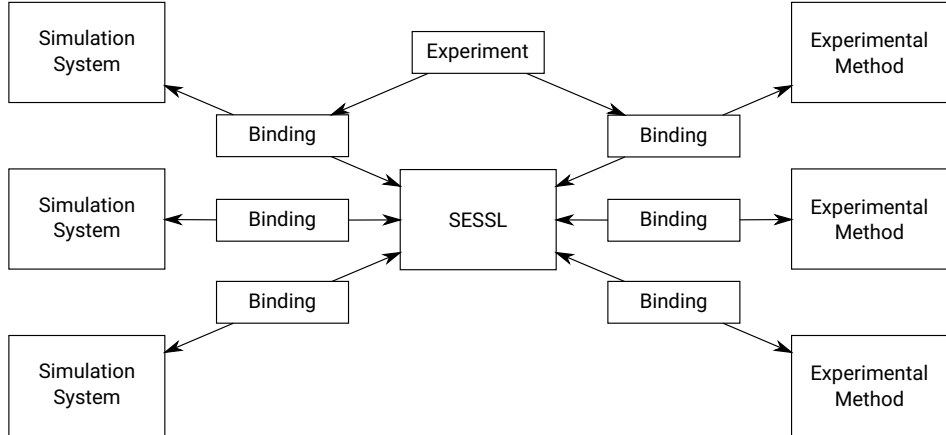
Discussion & Conclusions



cf. B. Zeigler et al. *Theory of Modeling and Simulation*. 2000







SESSL exploits Apache Maven² for project management. It allows

- declaring dependencies between components
- publishing artifacts to public repositories
- downloading needed dependencies from public repositories³
- bundling experiments in an executable package for replication

Replicating a packaged experiment is straightforward (one-click).

Artifact Review and Badging in the ACM Digital Library (e.g., TOMACS, PADS)⁴



²<https://maven.apache.org>

³<https://search.maven.org>

⁴<https://www.acm.org/publications/policies/artifact-review-badging>

SESSL provides an **internal domain-specific language**. A SESSL experiment only consists of valid Scala⁵ code. Consequently, SESSL experiments are **executable** like any other Scala program.

SESSL experiments can be augmented with arbitrary Scala code. Thus, SESSL experiments are **flexible** and can be extended with custom features ad hoc.

Scala runs on the Java Virtual Machine (JVM), which is available for every major computer architecture and operating system. Most computers have a JVM installed. Thus, SESSL experiments are **platform-independent**.

⁵<https://scala-lang.org>



```
import sessl._
import sessl.mlrules._

execute {

  new Experiment with Observation with CSVOutput {

    model = "./prey-predator.mlrj"
    simulator = SimpleSimulator()

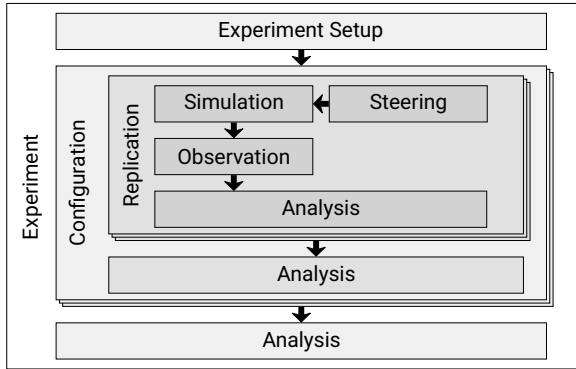
    stopTime = 100

    replications = 5

    scan("wolfGrowth" <- (0.0001, 0.0002))

    observe("s" ~ count("Sheep"))
    observeAt(range(0, 1, 100))

    withRunResult(writeCSV)
  }
}
```



```
new Experiment {
    // ...
    stopCondition = // ...
    replicationCondition = // ...
    observe(/* ... */)
    observeAt(/* ... */)

    withRunResult {
        // ...
    }
    withReplicationsResult {
        // ...
    }
    withExperimentResult {
        // ...
    }
}
```

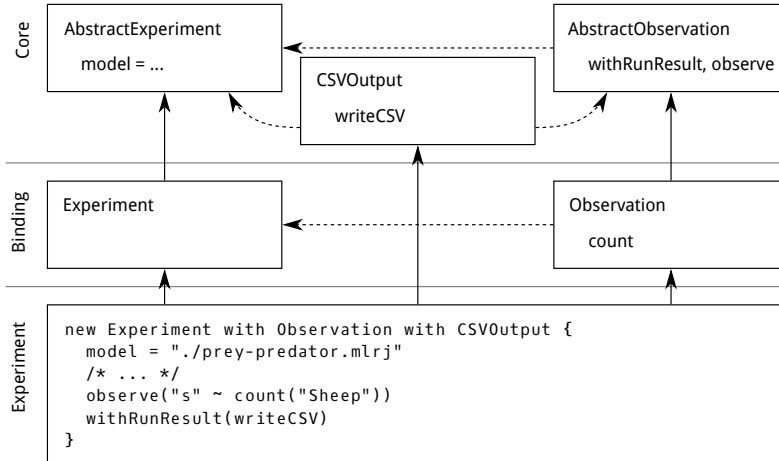
cf. S. Rybacki et al. "Template and Frame Based Experiment Workflows in Modeling and Simulation Software with WORMS". In: *IEEE SERVICES*. 2012. DOI: [10.1109/SERVICES.2012.22](https://doi.org/10.1109/SERVICES.2012.22)



Code is organized in **traits**.

- A binding to an external software X contains code that is specific to X .
- The SESSL core contains code that is not specific to external software and can be reused across bindings.
- A concrete Experiment **mixes in** the traits with the required features:

```
new Experiment with Observation with CSVOutput
```



Key Design Concepts

Playing around with a simple simulation experiment

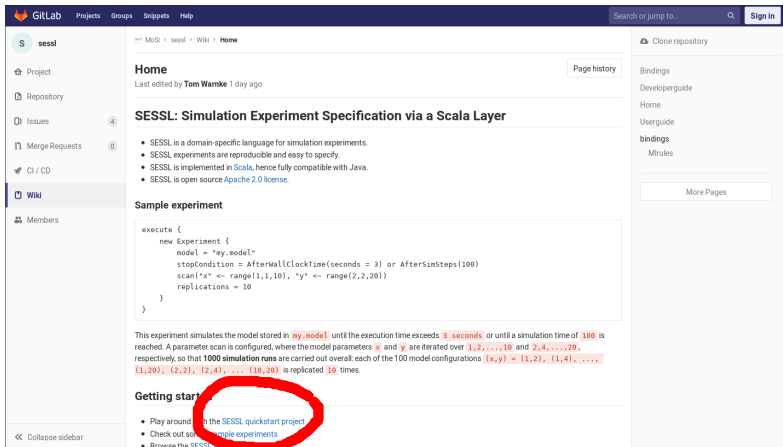
Complex simulation experiments

Extending SESSL

Discussion & Conclusions



<http://sessl.org>



The screenshot shows the GitLab interface for the SESSL project. The left sidebar contains navigation links: Project, Repository, Issues (4), Merge Requests (0), CI / CD, Wiki (selected), and Members. The main content area is titled 'Home' and 'SESSL: Simulation Experiment Specification via a Scala Layer'. It lists key features: SESSL is a domain-specific language for simulation experiments, experiments are reproducible and easy to specify, it is implemented in Scala (hence fully compatible with Java), and it is open source under the Apache 2.0 license. A 'Sample experiment' section shows a Scala code snippet for an experiment. Below the code, a paragraph describes the simulation parameters and iterations. The 'Getting started' section is partially visible, with a red circle highlighting the first bullet point: 'Play around with the SESSL quickstart project'.

Home
Last edited by **Tom Warnke** 1 day ago

SESSL: Simulation Experiment Specification via a Scala Layer

- SESSL is a domain-specific language for simulation experiments.
- SESSL experiments are reproducible and easy to specify.
- SESSL is implemented in [Scala](#), hence fully compatible with Java.
- SESSL is open source [Apache 2.0 license](#).

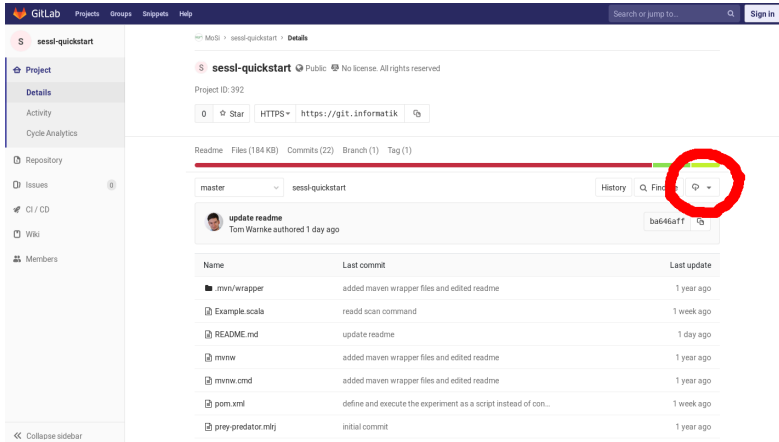
Sample experiment

```
execute {
  new Experiment {
    model = "my.model"
    stopCondition = AfterWallClockTime(seconds = 3) or AfterSimSteps(100)
    scan("x" <- range(1,1,10), "y" <- range(2,2,20))
    replications = 10
  }
}
```

This experiment simulates the model stored in `my.model`, until the execution time exceeds `3 seconds` or until a simulation time of `100` is reached. A parameter scan is configured, where the model parameters `x` and `y` are iterated over `1,2,...,10` and `2,4,...,20`, respectively, so that **1000 simulation runs** are carried out overall: each of the 100 model configurations `(x,y) = (1,2), (1,4), ..., (1,20), (2,2), (2,4), ..., (10,20)` is replicated **10** times.

Getting started

- Play around with the **SESSL quickstart project**
- Check out some [sample experiments](#)
- Browse the [SESSL documentation](#)



The screenshot shows the GitLab interface for the 'sessl-quickstart' project. The left sidebar contains navigation links: Project, Details, Activity, Cycle Analytics, Repository, Issues (0), CI / CD, Wiki, and Members. The main content area shows the project details for 'sessl-quickstart', which is public and has no license. It displays the project ID (392), a star count (0), and the HTTPS URL (https://git.informatik). Below this, there's a progress bar and a dropdown menu for the 'master' branch. A red circle highlights the 'clone' button (a square with a downward arrow) next to the 'Find' button. Below the clone button, there's a commit entry for 'update readme' by Tom Warnke, authored 1 day ago, with the commit hash 'ba646a1ff'. At the bottom, a table lists the files in the repository with their last commit and update dates.

Name	Last commit	Last update
■ .mvn/wrapper	added maven wrapper files and edited readme	1 year ago
📄 Example.scala	readd scan command	1 week ago
📄 README.md	update readme	1 day ago
📄 mvnw	added maven wrapper files and edited readme	1 year ago
📄 mvnw.cmd	added maven wrapper files and edited readme	1 year ago
📄 pom.xml	define and execute the experiment as a script instead of con...	1 week ago
📄 prey-predator.mlfj	initial commit	1 year ago



Experiment.scala



model.mlrj



pom.xml

Experiment The experiment is specified in a Scala file.

Model Typically, the model to simulate is also defined in a file.

Maven file This contains information about software dependencies.

The other files are for convenience.



The quickstart project packages an experiment with a simple prey-predator model defined in ML-Rules⁶.

- Initially, the model contains 1000 sheep and 1000 wolves.
- Three concurrent processes produce the model behavior:
 - Sheep -> 2 Sheep @ sheepGrowth;
 - Sheep + Wolf -> 2 Wolf @ wolfGrowth;
 - Wolf -> @ wolfDeath;
- Each process is equipped with a rate.

⁶C. Maus et al. "Rule-based multi-level modeling of cell biological systems". In: *BMC Systems Biology* 5.1 (2011). DOI: [10.1186/1752-0509-5-166](https://doi.org/10.1186/1752-0509-5-166).



Hands-on:

- (If necessary) **install a JRE and/or set the environment variable `$JAVA_HOME`**
- Inspect the Maven file
- Inspect the experiment file
- Run the experiment with `run.sh` or `run.bat`
- Plot the results of a run (e.g., with Excel or R)
- Add observation of the predator species (`wolf`)
- Change the `wolfGrowth` parameter to 0
- Adapt the stop condition to stop the simulation if it runs too long

Key Design Concepts

Playing around with a simple simulation experiment

Complex simulation experiments

Extending SESSL

Discussion & Conclusions

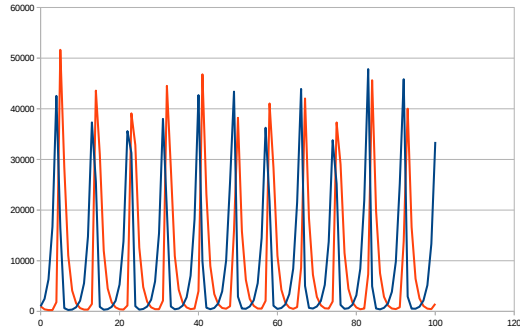


We might want to determine the time points at which more wolves than sheep were observed in each run.

We can write arbitrary Scala code in the analysis blocks and process the accumulated simulation results.

Such post-processing can support the exploratory analysis of the experiment results with other tools (e.g., R). It can also be packaged with an experiment for **replicable output analysis**.

```
new Experiment with Observation {  
  // ...  
  withRunResult { result =>  
    val sheep = result.trajectory[Double]("s").toMap  
    val wolves = result.trajectory[Double]("w").toMap  
  
    // determine the time points with more wolves than sheep  
    val moreWolves = sheep.keys.filter(t => sheep(t) < wolves(t)).toSeq.sorted  
    println(moreWolves.mkString(", "))  
  }  
}
```



5.0, 6.0, 7.0, 8.0, 9.0, 14.0, 15.0, 16.0, 17.0, 18.0, ...



Design of Experiments: systematic exploration of the model's parameter space⁷

For example:

- Factorial Design
- Central Composite Design
- Latin Hypercube Design

⁷S. M. Sanchez and H. Wan. "Work smarter, not harder: A tutorial on designing and conducting simulation experiments". In: *WSC 2015*. DOI: [10.1109/WSC.2015.7408296](https://doi.org/10.1109/WSC.2015.7408296).

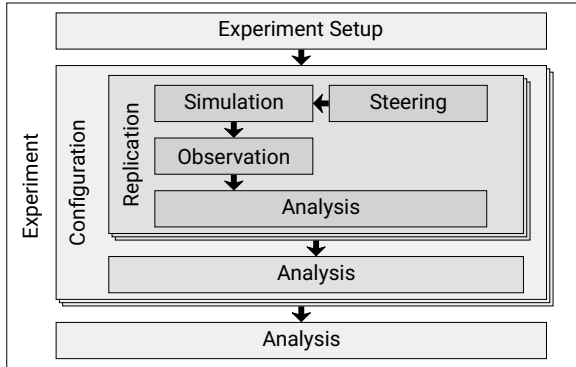
Assume a model with input parameters a, b, \dots, g . The following snippet produces $1 \times 2 \times 10 \times 5 \times 9 = 900$ design points.

```
execute {  
  new Experiment with CentralCompositeDesign with LHCSampling8 {  
    // ...  
    set("a" <- 1.0)  
    scan("b" <- ("on", "off"))  
    scan("c" <- range(0.1, 0.1, 1.0))  
    lhc(numPoints = 5)("d" <- interval(0, 5), "e" <- interval(0.0, 10.0))  
    centralComposite("f" <- interval(1, 2), "g" <- interval(3, 4))  
  }  
}
```

⁸LHCSampling is implemented in a **binding** to SSJ (P. L'Ecuyer et al. "SSJ: A Framework for Stochastic Simulation in Java". In: WSC 2002. DOI: [10.1109/WSC.2002.1172890](https://doi.org/10.1109/WSC.2002.1172890))

SESSL dynamically determine how many replications are necessary. It executes runs in batches and checks the given replication condition after each batch. This is done for each parameter combination.

For instance, we might want to execute replications until the confidence interval (CI) of the observable's mean is small enough. Here, small enough means that the 99 % CI's half-width is less than 1 % of the mean.



```
new Experiment {
    // ...
    stopCondition = // ...
    replicationCondition = // ...
    observe(/* ... */)
    observeAt(/* ... */)

    withRunResult {
        // ...
    }
    withReplicationsResult {
        // ...
    }
    withExperimentResult {
        // ...
    }
}
```

For each parameter combination, execute replications until the 99% CI for the mean of all observations of a is small enough, but at most 1000 replications.

```
execute {
  new Experiment with Observation with ParallelExecution {
    // ...
    parallelThreads = -1

    stopTime = 100

    observe("a" ~ count("A"))
    observeAt(stopTime)

    replicationCondition = MeanConfidenceReached(
      confidence = 0.99,
      relativeHalfWidth = 0.01,
      observable = "a" or FixedNumber(1000)
    )
    batchSize = 3
  }
}
```

An experiment is executed to evaluate the target function of an optimization algorithm.

For example, a meta-heuristic optimization algorithm:

- Evolutionary Algorithms
- Simulated Annealing
- Particle Swarm Optimization
- Differential Evolution

```

minimize { (params, objective) => execute {
  new Experiment with Observation with ParallelExecution {
    //...
    for ((input, value) <- params.values)
      set(input <- value)

    withExperimentResult { results =>
      objective <- // ... calculate target function value from experiment results
    }
  }
} } using new Opt4JSetup9 {
  param(name = "a", lowerBound = 5E-5, upperBound = 5E-3)
  // ...
  optimizer = ParticleSwarmOptimization(iterations = 30, particles = 20)
  withOptimizationResults { results => println("Overall results: " + results.head) }
}

```

⁹Implemented in a [binding](#) to Opt4J (M. Lukasiwycz et al. "Opt4J - A Modular Framework for Meta-heuristic Optimization". In: *GECCO 2011*. 2011. DOI: [10.1145/2001576.2001808](#))

Statistical model-checking is a method to decide whether a random simulation run of a model satisfies a given **formal property** with at least a certain probability¹⁰. The statistical parameters are:

- the probability threshold p
- the probability bounds for Type I and Type II errors α and β , and
- the width of the indifference region δ .

The property is usually defined in some temporal logic. For example, SESSL supports the Metric Interval Temporal Logic (MITL)¹¹.

By defining validation criteria as properties, **replicable validation experiments** can be packaged and published together with a model.

¹⁰G. Agha and K. Palmkog. "A Survey of Statistical Model Checking". In: *ACM TOMACS* 28.1 (2018). DOI: [10.1145/3158668](https://doi.org/10.1145/3158668).

¹¹O. Maler and D. Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. 2004. DOI: [10.1007/978-3-540-30206-3_12](https://doi.org/10.1007/978-3-540-30206-3_12).



```
execute {
  new Experiment with Observation with StatisticalModelChecking {
    //...
    test = SequentialProbabilityRatioTest( // these parameters yield 180 simulation runs
      p = 0.8,
      alpha = 0.05,
      beta = 0.05,
      delta = 0.05)

    // The following MITL property states that sheep do not die out the first 50 time units.
    prop = MITL(G(0, 50)(OutVar[Double]("s") > Constant(0)))
  }
}
```


Key Design Concepts

Playing around with a simple simulation experiment

Complex simulation experiments

Extending SESSL

Discussion & Conclusions



Assume we want to import a CSV file with a list of design points for an experiment.

```
sheepGrowth,wolfGrowth,wolfDeath
0.1,0.1,0.1
0.0,0.1,0.1
0.1,0.0,0.1
0.1,0.1,0.0
```

Step 1: Develop the code to read in the file directly in the experiment.

```
new Experiment with Observation with CSVOutput {
  model = "./prey-predator.mlrj"
  simulator = SimpleSimulator()
  // ...
  val designPoints = ??? // invoke CSV reader library (omitted)
  configs(designPoints.toList: _) // use imported design points as model configurations
}
```

Step 2: Create a trait and wrap the code in a reusable method.

```
trait CSVInput {
  this: AbstractExperiment =>

  def designFromCSV(fileName: String): Unit = {
    val designPoints = ??? // invoke CSV reader library (omitted)
    configs(designPoints.toList: _*) // use imported design points as model configurations
  }
}
```

Step 3: Use the new trait and the defined method in future experiments.

```
execute {
  new Experiment with Observation with CSVOutput with CSVInput {
    model = "./prey-predator.mlrj"
    simulator = SimpleSimulator()
    // ...

    designFromCSV("design.csv")
  }
}
```

Developing a new binding to a simulation system requires

- an API exposed by the simulation system
- some knowledge of SESSL's architecture.

To make most of the features of SESSL available, two methods need to be implemented in the binding:

- Start a simulation run and wait for it to finish, for example
 - Start a process (e.g., via a command line interface)
 - Call an API method
- Convert the observed results to SESSL's result format, for example
 - Read in files
 - Retrieve stored observations via an API

For example, the binding for **PSSALib**¹² (written in C++) uses a CLI:

Start simulation runs:

```
pssa_cli/simulator
```

```
-m spdm
```

```
-i model.xml
```

```
-n 100
```

```
--tend 10000
```

```
--dt 10
```

```
-o <output-path>
```

Convert results to CSV:

```
pssa_cli/analyzer
```

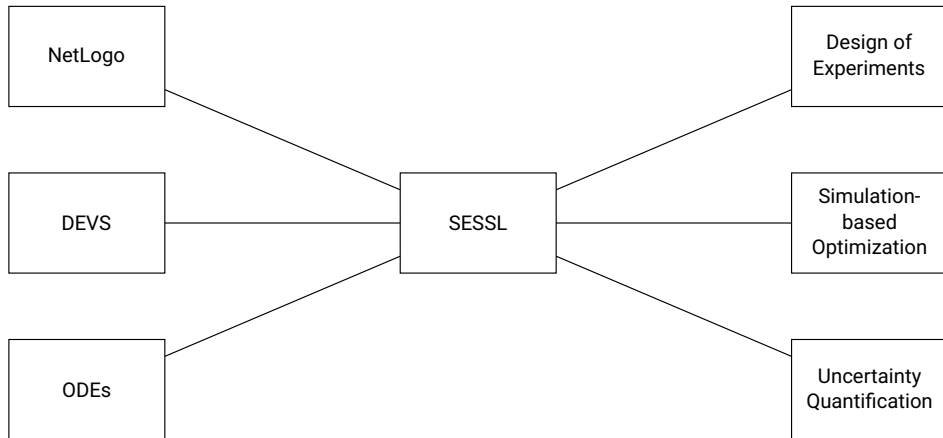
```
-r trajectories
```

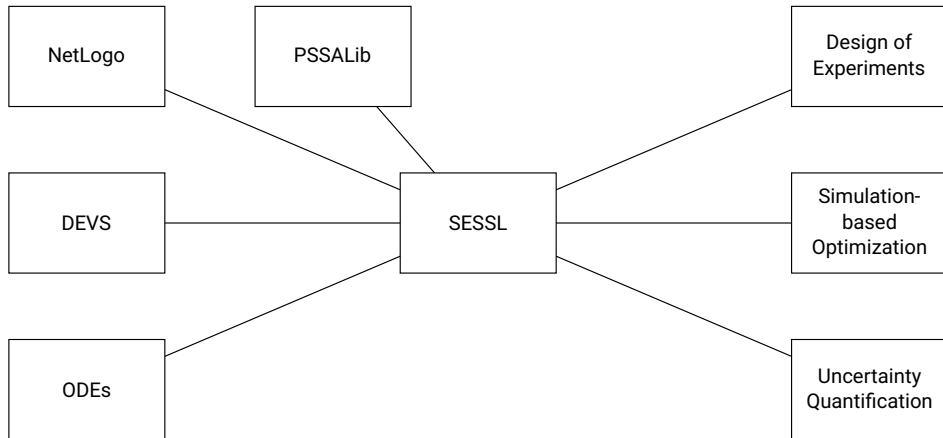
```
-i <input-path>
```

```
-s obs1,obs2
```

```
-o <output-path>
```

¹²Oleksandr Ostrenko et al. “pSSALib: The Partial-propensity Stochastic Chemical Network Simulator”. In: *PLOS Computational Biology* 13.12 (2017). DOI: [10.1371/journal.pcbi.1005865](https://doi.org/10.1371/journal.pcbi.1005865).





Key Design Concepts

Playing around with a simple simulation experiment

Complex simulation experiments

Extending SESSL

Discussion & Conclusions

Scala is a very powerful language, but **with great power comes great responsibility**.

To fully exploit SESSL and all its features, one must know Scala. Sometimes, the Scala embedding tends to provide **too much flexibility and power**.

For example, all these lines are valid Scala code and do the same thing:

```
withRunResult(writeCSV)
withRunResult(r => writeCSV(r))
withRunResult(writeCSV(_))
withRunResult(writeCSV _)
withRunResult { writeCSV }
withRunResult { r => writeCSV(r) }
withRunResult { writeCSV(_) }
withRunResult { writeCSV _ }
```

- Users may be overwhelmed by the possibilities of SESSL
- Experiment specifications with a lot of Scala code may be less readable

By using native Scala variables to store observables, type information about observables is usable by the Scala compiler. For example, the compiler can spot attempts to calculate a mean of non-numeric observations.

```
observe("s" ~ count("Sheep")) // "s" is a String value, information from count is lost
withExperimentResult(result =>
  println(result.mean[Double]("s")) // user needs to explicitly annotate the type Double
)
```

```
val s = observe(count("Sheep")) // s is of type Observable[Double], inferred from count
withExperimentResult(result =>
  println(result.mean(s)) // the compiler allows the call to mean without annotations
)
```

This exemplifies the **trade-off** between exploiting features of the host language Scala and “protecting” users from having to learn or understand Scala.

Maven makes replicating simulation experiments very easy, if:

- all dependencies run on the JVM
- all dependencies are available from a public Maven repository

This is true for a lot of software.

However, natively compiled software (e.g., written in C++) can not be managed by Maven and requires manual setup by the user (only once).

Alternatively, SESSL experiments and pre-compiled software can be bundled together in a **Docker container** or similar architecture-agnostic packages.



SESSL is made for first defining experiments with a DSL and then running them. A more **interactive, explorative experimentation style** requires a different approach, for example integration in a Read-Eval-Print-Loop (REPL) as in R.

As SESSL experiments are just plain Scala code, the Scala REPL can be used for interactive experimentation. Alternatively, Scala experiments can be started from external tools (e.g., Python scripts).



- SESSL is modular and easily allows
 - creating bindings to external software
 - adding new features
 - composing features for experiments
- SESSL makes experiment replication as easy as possible
- SESSL can be combined with external software and custom ad hoc implementations in various ways
- SESSL facilitates wrapping code in reusable chunks
- SESSL development relies on standard programming tools (IDEs, libraries)
- SESSL experiments rely on persistent artifacts and will never* stop working