Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

COMPLEX SIMULATION EXPERIMENTS MADE EASY

Tom Warnke
Adelinde M. Uhrmacher

Institute of Computer Science
University of Rostock
Albert-Einstein-Straße 22
18059 Rostock, GERMANY

ABSTRACT

Diverse methods for complex simulation experiments can contribute to developing and gaining insights into simulation models, for example simulation-based optimization, sensitivity analysis, or statistical model-checking. An effective tool for conducting simulation experiments must be highly flexible to support such a broad range of experimental methods. Furthermore, to facilitate reproducibility and communication of simulation experiments, an effective tool for simulation experimentation must yield experiment descriptions that are easily portable, executable, and human-readable. In this tutorial we introduce SESSL, a domain-specific language for setting up simulation experiments. SESSL is flexible and extensible, and experiment descriptions are executable, often succinct, and can be executed reproducibly across machines and operating systems. Based on a few examples, we demonstrate how SESSL can be leveraged to easily conduct complex simulation experiments while reusing existing software and methods, and how SESSL's capabilities can be extended and combined with arbitrary simulation software via bindings.

1 INTRODUCTION

Diverse types of complex simulation experiments can contribute to developing simulation models. For example, model input parameters can be tuned via simulation-based optimization; the parameters for which additional data shall be gathered can be determined via sensitivity analysis; or the model can be checked against formally defined properties via statistical model-checking. Thus, simulation experiments are central ingredients to calibrating and validating a model (Leye et al. 2009). But even after validation, simulation models are experimented with. For example, a model of a supply chain might be used in simulation experiments to explore the performance of alternative supply chain designs (Persson and Olhager 2002).

Simple simulation experiments can often be implemented in a general-purpose programming language. For example, a simulation experiment exploring the behavior of a model that depends on two parameters can be easily implemented with two nested loops:

```
1 for (x <- 1 to 5)
2   for (y <- List(1, 10, 100, 1000))
3     runSimulation(param1 = x, param2 = y)
```

However, more complex experiments quickly lead to program code that is hard to read as well as hard to maintain or even reuse. The simulation experiment that is implemented by nested loops, for example, requires one loop for each considered parameter. Consequently, the code's readability decreases with the number of parameters to sweep, and even more so with more complex experiment setups, possibly including code for managing the observations for each parameter combination etc.

Alternatively, some simulation tools provide built-in experimentation capabilities. For example, NetLogo's BehaviorSpace tool allows for configuring simulation experiments in a graphical user interface or in an

XML format (Wilensky 2018). Similarly, Repast Simphony provides a batch run dialog to create simulation experiments executing various model parameterizations (North et al. 2006). Complex experiments beyond simple parameter sweeps, however, are not possible with such approaches.

These considerations lead to two central requirements that an effective tool for simulation experimentation must satisfy. First, the tool must be sufficiently flexible to support a wide range of experiments. Second, the tool must work with experiment descriptions that are compact, declarative, and can be easily understood by users. A third requirement arises from the nature of experiments according to the scientific method: experiments must be reproducible. This means that given the description of a simulation experiment, it must be possible to execute the experiment again and obtain the same results. Considering how complex many simulation programs are and how they depend on other software (or even hardware), reproducibility is a requirement that is surprisingly hard to satisfy.

In this tutorial, we introduce the Simulation Experiment Specification via a Scala Layer (SESSL), a domain-specific language for specifying and executing simulation experiments (Ewald and Uhrmacher 2014). SESSL is embedded in Scala, and SESSL experiments are executable Scala programs. However, due to SESSL's language design, experiment descriptions have a declarative core and are typically succinct and readable. SESSL can be easily extended and combined with Scala code, making it highly flexible. Running on the Java Virtual Machine (JVM) with automated dependency management, SESSL experiments can be shared and reproduced on all common operating systems. Thus, SESSL satisfies all three defined requirements.

2 OVERVIEW OF SESSL

SESSL has been designed as a lightweight tool that mainly connects external software. By using SESSL as a common interface for configuring and executing experiments, existing software (e.g., for determining experiment designs, executing simulation runs, or analyzing simulation output) can be composed easily. The communication between SESSL and the external software is managed by *bindings*. A binding can connect SESSL to external software via various types of interfaces, for example by reading and writing input and output files, starting processes, or invoking an application programming interface (API). This way, any external software that provides some kind of API, command line interface, or similar interface can be connected to SESSL, regardless of the programming language or framework used.

In a simulation experiment, the binding for the simulation system that executes the simulation runs can be combined with other bindings and software. This leads to one of the main advantages of SESSL: As bindings can be freely combined, a once implemented experimental method, for example simulation-based optimization, can be applied to all simulation systems for which a binding exists. Additionally, even more complex experiment setups can be realized via composing bindings. For example, with bindings to several simulation systems that can simulate the same model, the results of simulation runs as well as their performance can be easily compared. Similarly, two consecutive experiments can be defined, where the first experiment uses simulation-based optimization to determine an optimal parametrization of the model regarding some target function, and in a second experiment a local sensitivity analysis around the previously identified optimum is conducted to quantify the robustness of the optimum.

In the following, we demonstrate how different types of simulation experiments can be defined and executed with SESSL. We start with a short description of SESSL's implementation principles, using a simple experiment as an example. Subsequently, we show

- how to create model configurations from an experiment design
- how to use complex stop and replication conditions
- how to aggregate and output simulation results
- how to apply a simulation-based optimization (with a binding for Opt4j)
- how to conduct a sensitivity analysis with a simulation model
- how to analyze the behavior of a model with statistical model-checking

Executable example experiments with these and other features of SESSL, such as calulating a meta-model from simulation output or specifying observations of simulation runs can be downloaded at https://git.informatik.uni-rostock.de/mosi/sessl-examples.

Finally, we demonstrate how SESSL can be extended. We show how we added the ability to read in experiment designs from a CSV file into a SESSL experiment. We also sketch the steps for making the features of SESSL available for a simulation system by creating a new binding. A discussion of the design principles of SESSL concludes the paper.

2.1 Implementation Principles

SESSL experiment specifications are valid Scala code. Consequently, the infrastructure that is available for working with Scala can be reused for SESSL. An Integrated Development Environment (IDE) with capabilities for editing Scala code, for example, will flag syntactical errors in SESSL experiment specifications. To execute a SESSL experiment, again an IDE is the recommended way. Alternatively, a SESSL experiment is typically accompanied by executable scripts to run the experiment on Unix and Windows systems. SESSL is published as open-source software under the Apache License, Version 2.0, at sessl.org.

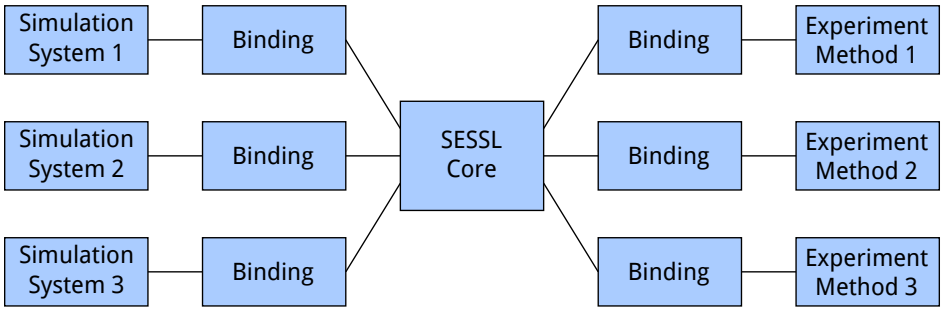

Figure 1: SESSL is structured into a core and bindings that connect the core to external software for simulation or experimental methods. In experiments, bindings can be freely combined.

SESSL is structured into a core and bindings (Figure 1). The core provides the common interfaces and basic features of experiment specifications. The bindings, on the other hand, primarily connect SESSL to external tools and libraries, translating between the interfaces defined in the core and the external software. Each binding implements the subset of SESSL features that is supported by the external software. Thus, the different feature sets of different simulation tools are reflected in the bindings while using the same interfaces. Besides bindings to external software, bindings are also used to package generic, but advanced experimental methods.

A concrete SESSL experiment always uses the core and at least one binding for a simulation system. The management of the dependencies required to execute an experiment is delegated to Apache Maven (maven.apache.org). The bindings to use are declared in `pom.xml`, a file that always accompanies a SESSL experiment description, enabling Maven to download all required artifacts. If the external software is available via Maven repositories (many Java-based software products are), it can be managed by Maven as well. Otherwise – for example if the external software is compiled to machine code – it has to be installed separately. Regardless of the simulation system used, the basic aspects of a SESSL experiment are the same (as shown in the next sections). This enables porting an existing SESSL experiment from one simulation system to another. If a simulation experiment attempts to use features that the simulation system at hand does not support, the compilation or execution of the experiment will fail, which gives quick feedback about the correctness of the experiment specification.

As a starting point for working with SESSL, a quickstart project with a simple experiment setup is available for download. The SESSL quickstart project uses a Maven wrapper (Takari 2010), which relieves the user of manually installing Maven. Thus, on a machine with an existing Java installation, the quickstart

```
1  object ExampleExperiment extends App {
2
3    import sessl._
4    import sessl.mlrules._
5
6    execute {
7      new Experiment with Observation with ParallelExecution with CSVOutput {
8        model = "./prey-predator.mlrj"
9        simulator = SimpleSimulator()
10       parallelThreads = -1
11
12       stopTime = 100
13
14       replications = 5
15
16       scan("wolfGrowth" <~ (0.0001, 0.0002))
17
18       observe("s" ~ count("Sheep"))
19       observeAt(range(0, 1, 100))
20
21       withRunResult(writeCSV)
22     }
23   }
24 }
```

Figure 2: The experiment specification in the SESSL quickstart project executes 5 replications each of 2 parametrizations of a Lotka-Volterra model defined in ML-Rules.

project can be downloaded and started without further manual installations. If the `pom.xml` is adapted, Maven automatically downloads new dependencies as needed.

2.2 A Simple Experiment

The quickstart project contains the experiment specification shown in Figure 2. In this simple experiment, the binding for the modeling and simulation tool ML-Rules (Maus et al. 2011) is used to simulate a Lotka-Volterra prey-predator model (Volterra 1926) with the `SimpleSimulator`, a specific discrete-event simulation algorithm in the ML-Rules package (Helms et al. 2017). Five runs are executed with the parameter "wolfGrowth" set to 0.0001, and five with 0.0002. Each run stops at simulation time 100, and is observed at every time point from 0 to 100 in steps of 1. For each observation time, the number of "Sheep" entities in the model is recorded. After each run, the observed trajectory is written to a CSV file. Although this experiment is very simple, it already exemplifies some of the most important aspects of SESSL. The example also shows that SESSL specifications consist of plain Scala code. We now walk through the code line by line.

Line 1 defines an `App` object, which is a Scala method to make the enclosed code executable, comparable to a `main` method. The `import` statements in lines 3 and 4 determine which features of SESSL and its bindings are available in the experiment. Here, the SESSL core and the binding for ML-Rules are loaded. As mentioned above, the core and the binding to use must also be declared in the `pom.xml`. In line 7, a new instance of the `Experiment` *class* is created with three additional *traits*: `Observation`, `ParallelExecution`, and `CSVOutput` (classes and traits are Scala concepts for object-oriented programming). The class `Experiment` is the central entity of SESSL and has to be defined in every binding that connects SESSL to a simulation system. Which `Experiment` class is used is governed by the `import` statements such as the one in line 4. Thus, the `Experiment` class in the ML-Rules binding is used here. The `Experiment` classes in the individual bindings implement the communication with the external simulation systems, but also reuse code from their

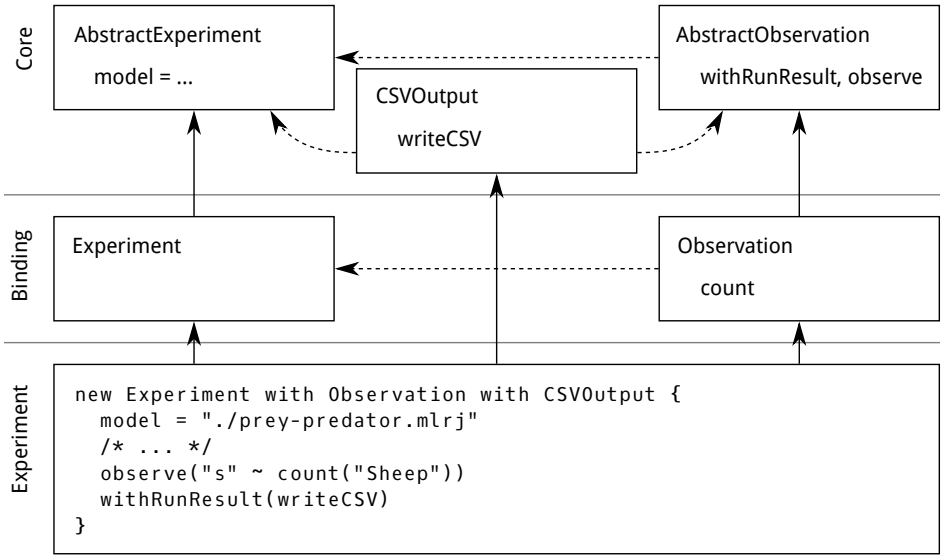

Figure 3: A specific experiment uses code from the core and the binding for the simulation system used. The diagram shows some classes and traits that are used in the example in Figure 2 as well as some of the methods they contain. Solid arrows show sub-type relations ("is a sub-type of"), dashed arrows show self-type relations ("can only be mixed in an instance of"). Thus, for example, the trait `CSVOutput` can only be used in experiments that are subtypes of `AbstractObservation` and `AbstractExperiment`, such as the example experiment. These dependencies are checked by the Scala compiler.

superclass `AbstractExperiment`, which is defined in the SESSL core. Similarly, the traits `Observation` and `ParallelExecution` are defined in the binding for ML-Rules and extend `AbstractObservation` and `AbstractParallelExecution`, which are located in the SESSL core. `CSVOutput`, on the other hand, is defined in the core and agnostic of specific simulation systems.

All the statements inside the `new Experiment with ... { ... }` block are executed as the constructor of the `Experiment` instance. The class `Experiment` provides only the most basic features, such as selecting a model to simulate (line 8), selecting the simulation algorithm (line 9), parametrizing the model (line 16), as well as setting stop (line 12) and replication conditions (line 14). Further features are provided by generic and simulation system-specific traits that can be *mixed in* when creating an experiment instance. For example, the `observe`, `observeAt`, and `withRunResult` statements in lines 18 and 19, and 21 are provided by the `Observation` trait. It extends the existing code in `AbstractObservation` with ML-Rules-specific observation code, which is used here to record observations of the count of the species `Sheep`. The `parallelThreads = -1` statement in line 10 is provided by the `ParallelExecution` trait and is translated to a configuration parameter for the ML-Rules simulation package, making it use all but one available CPU cores for executing simulation runs in parallel. Lastly, the `writeCSV` statement in line 21 is provided by the `CSVOutput` trait. Figure 3 illustrates the dependencies between the classes and traits.

This example shows that many aspects of SESSL experiments can be specified with declarative statements. Moreover, it demonstrates how experiments can be enriched with traits, and how traits from bindings and the SESSL core can be combined. By decomposing the implementations of experiment features into traits, the corresponding code can be developed and tested independently. It also ensures that the traits can be reused in experiments regardless of other traits that have been mixed in. Further, the SESSL core contains code that can be reused when implementing simulation-system-specific traits such as `Observation`. But, not only the code in SESSL's core or bindings can be reused. It is also possible to fix specific experiment aspects in a user-defined trait, which can then be reused to define experiments. This is useful, for example, to define many similar experiments without repeating all commands in every

```
object ExampleExperiment extends App {

  import sessl._
  import sessl.mlrules._

  trait PreyPredatorExperiment extends Experiment with ParallelExecution {
    model = "./prey-predator.mlrj"
    simulator = SimpleSimulator()
    stopTime = 100
    replications = 5
    parallelThreads = -1
  }

  execute {
    new Experiment with PreyPredatorExperiment with Observation with CSVOutput {
      scan("wolfGrowth" <~ (0.0001, 0.0002))

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

      withRunResult(writeCSV)
    }
  }
}
```

Figure 4: The experiment specification from Figure 2 after a user-defined trait has been factored out to make some commands of the experiment configuration reusable.

experiment specification. In the experiment in the quickstart project (Figure 2), some commands can be factored out into a separate trait, which yields code that facilitates reuse of certain experiment aspects (Figure 4).

Similar to the SESSL quickstart project, SESSL experiments can be easily packaged and shared. Executing a packaged SESSL experiment and reproducing its results is straightforward. However, to yield exactly the same results, the random number generation algorithm that produces the random numbers as well as its seed need to be configured as well. We omit this here for brevity.

3 COMPLEX EXPERIMENTS

After introducing bindings and traits as SESSL's core concepts for defining experiments, we now show how complex, non-trivial simulation experiments can be specified with SESSL. For space reasons, we will in most cases only show snippets of each experiment. The full experiment specifications are available for download at https://git.informatik.uni-rostock.de/mosi/sessl-examples.

3.1 Experiment Design

Many simulation experiments vary the input parameters of the model in some way. The methods to systematically do so are subsumed under the term "experiment design" or "design of experiments" (Sanchez 2005). SESSL provides some generic, simulation system-agnostic features to create a set of model configurations from user input.

```
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))
```

The code in this listing yields 900 parameter configurations by crossing 1 fixed value for `a`, 2 values for `b`, a range of 10 values $(0.1, 0.2, \ldots, 1.0)$ for `c`, 5 points in a randomized Latin Hypercube design for `d` and `e`, and 9 points in a 2D central composite design for `f` and `g`. To construct a Latin Hypercube or central composite design, the intervals for each parameter can be given. Both of these more complex experiment design methods are implemented in specific traits. In Section 4.1, we additionally show how a trait to import an experiment design from a CSV file can be implemented.

3.2 Stop and Replication Conditions

Configuring dynamic replication and stop conditions rather than fixed replication numbers and stop times is especially valuable for stochastic models.

SESSL enables complex stop conditions in two ways. First, the stop conditions defined by a simulation system can be adopted in the SESSL binding and then used in experiment specifications. Second, if the simulation system provides corresponding hooks, SESSL's nature as an internal domain-specific language facilitates defining predicates as Scala code to be injected as stop conditions. However, this implementation variant usually requires simulator-specific expressions in the experiment specification, which are often verbose and hard to read. A strategy to mitigate this problem is to wrap recurring boilerplate code in methods in a simulation-system-specific trait. In general, the evolution of SESSL is often driven by identifying recurring patterns in experiment specifications and extracting common code chunks into reusable methods or traits.

```
stopCondition = Custom((state, time) => state.getAgentsAlive.isEmpty || time >= 100)
```

The listing above shows a stop condition for an experiment with the SESSL binding for ML3, an agent-based modeling language (Warnke et al. 2015). In this experiment, each simulation run stops as soon as all agents are dead or the simulation time exceeds 100. It is also possible to integrate implementations of steady state detection methods, such as Schruben (1983).

The code for replication conditions is typically easily reusable, as external simulation systems are invoked by SESSL on a per-run basis. Thus, replication conditions can be evaluated based on the recorded observation of already completed runs, after the simulation-system-specific observation logic has translated the simulation output to simulation-system-independent observations. A common example for such a replication condition is constraining the width of a confidence interval. In the following listing, a 99% confidence interval of the mean value of the observed variable `a` over all replications is calculated. More replications are executed until the relative half-width of the confidence interval is less than 1% or the number of replications has reached $1,000$. Replications are executed in batches of 10.

```
replicationCondition = MeanConfidenceReached(
  confidence = 0.99,
  relativeHalfWidth = 0.01,
  varName = "a") or FixedNumber(1000)

batchSize = 10
```

3.3 Result Processing

Whereas in practice, the results of complex simulation experiments are typically processed using dedicated tools such as R, SESSL provides some features to post-process the results of a simulation experiment. With statements such as `withRunResult{...}`, SESSL allows for injecting functions that are invoked on the results of every run, the runs of every configuration, or all simulation runs of the entire experiment. These functions can be predefined (which is natural through Scala's support of functional programming). The trait `CSVOutput` that was introduced in the listing in Figure 2, for example, provides the predefined functions `writeCSV`, which can be applied to results of a run, results of replications, or results of an entire experiment. In all cases, the recorded observations are written to standard-conforming CSV files

in a structure of directories that also contain the configuration of the corresponding run. This way, the simulation experiment results can be further processed and analyzed with R, Excel, or other tools.

It is also possible to define ad-hoc functions for processing results. As we will see in the next sections, this is vital for more complex experiment setups where the execution and evaluation of simulation runs are embedded in a higher-level algorithm. But, provided that the user is sufficiently proficient in Scala, this can also be useful to analyze the simulation results directly in the experiment specification. For example, based on the experiment with the prey-predator model in Figure 2, the following code can be used to obtain all time points at which more predator than prey entities were observed:

```
1 withRunResult { result =>
2   val sheep = result.trajectory[Double]("s").toMap
3   val wolfs = result.trajectory[Double]("w").toMap
4   val moreWolfs = sheep.keys.filter(t => sheep(t) < wolfs(t)).toSeq.sorted
5 }
```

3.4 Simulation-based Optimization

Simulation-based optimization helps to calibrate a model towards a specific output or behavior by systematically tuning the model parameters (Moles et al. 2003). This requires a simulation experiment that yields the output for a given model configuration via selecting a simulator, stop conditions, observation times, and observables, etc., which we have already demonstrated in SESSL. Additionally, an optimization algorithm as well as a target function must be supplied.

Through a binding for the open-source library Opt4J (Lukasiewycz et al. 2011), SESSL gains access to implementations of several optimization algorithms. SESSL `Experiment` instances are integrated with these implementations by wrapping the experiment execution in an anonymous function, as shown in the following listing:

```
1  minimize { (params, objective) =>
2    execute {
3      new Experiment with /* experiment traits */ {
4        /* experiment setup */
5
6        for ((input, value) <- params.values)
7          set(input <~ value)
8
9        withExperimentResult { results =>
10         objective <~ /* calculate objective */
11       }
12     }
13   }
14 } using new Opt4JSetup {
15   /* optimizer setup */
16 }
```

The `minimize` block wraps an anonymous function that is defined in lines 1–14, which takes two inputs, `params` and `objective`, and executes the experiment defined in lines 3–12 when invoked. When the experiment is executed, the parameters in `params` are set as model parameters (lines 6–7) and the `objective` is set based on the results of the experiment (line 10). Thus, the anonymous function encodes evaluating a target function by executing a simulation experiment. This target function is repeatedly invoked by an algorithm that is wrapped in the Opt4J setup in lines 14–16.

As an example for a concrete target function, we show how the distance of the model output to a given reference trajectory can be minimized. This can be used to calibrate a model towards producing an observation of the modeled system. For example, a model of a biochemical signaling pathway can be tuned to reproduce some wet-lab measurements taken at specific time points, represented in Scala as follows:

```
val ref: Trajectory[Double] = List((  0, 5282),
                                   (120, 7561),
                                   (240, 8247),
                                   (360, 7772),
                                   (480, 7918),
                                   (600, 7814),
                                   (720, 7702))
```

Using these values as a reference, the target function can be defined based on simulation observations, as shown in the following listing. The distance between a trajectory produced by the simulation and the reference trajectory is calculated as a root-mean-square error (line 4, using the auxiliary function `Misc.rmse` from the SESSL core). The value of the target function is then the mean of all errors calculated for the individual runs (line 6).

```
withExperimentResult { results =>
  val distances = for (run <- results.runs) yield {
    val trajectory = run.trajectory[Double]("simOutput")
    Misc.rmse(trajectory, ref)
  }
  objective <~ distances.sum / distances.size
}
```

Opt4J is currently the only implementation for simulation-based optimization, but the approach is independent of the optimization algorithm used. Other external libraries or custom implementations can be integrated straightforwardly, again by reusing much of the existing code. For example, for each optimization algorithm, the parameters to vary and the ranges in which to vary them must be given. Opt4J-specific code, such as the selection of the optimization algorithm implementation, are wrapped in the SESSL binding for Opt4J. The optimizer setup also allows for defining result handling functions in a similar way as the experiment itself. In the following listing, lines 7–9 state that the final results of the optimization process shall be the output. This yields the optimal parameter combination found as well as the corresponding minimum of the target function.

```
} using new Opt4JSetup {
  param("diss", 5E-5, 5E-3)
  param("dephos", 0.01, 0.1)
  param("phos", 0.1, 10)
  param("syn", 0.1, 5)
  optimizer = ParticleSwarmOptimization(iterations = 30, particles = 20)
  withOptimizationResults { results =>
    println("Overall results: " + results.head)
  }
}
```

3.5 Sensitivity Analysis

Sensitivity analysis with SESSL is conducted in a similar way as simulation-based optimization. Again, the execution of the `Experiment` is wrapped in an anonymous function that serves as the target function for the enclosing block.

```
analyze { (params, objective) =>
  execute {
    new Experiment with /* experiment traits */ {
      /* experiment setup */

      for ((input, value) <- params.values)
        set(input <~ value)

```

```
      withExperimentResult { results =>
        objective <~ /* calculate objective */
      }
    }
  }
} using new TwoLevelFullFactorialSetup {
  /* sensitivity analysis setup */
}
```

The implementation of sensitivity analysis is factored out into the binding `sessl-analysis`, which also contains an implementation for conducting bifurcation analysis. Currently, the integration of more sensitivity analysis methods is still work in progress. However, the actual implementation work to add an existing sensitivity analysis to the methods available in SESSL is small, as a lot of the existing code can be reused.

3.6 Statistical Model-Checking

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 probability (Agha and Palmskog 2018). It is a useful technique to define validation experiments by formulating validity conditions as formal properties that can be checked on each simulation run. Typically, these properties are expressed in temporal logic. Currently, SESSL provides an implementation of the Metric Interval Temporal Logic (MITL) (Maler and Nickovic 2004) to specify properties. Compared to other temporal logic, MITL is especially suitable to express quantitative properties of trajectories. For example, for the prey-predator model a statistical model-checking experiment could verify that, in a random simulation run, the `Sheep` species does not die out during the simulation run. In SESSL's implementation of MITL, this could be expressed as:

```
prop = MITL(G(0, stopTime)(OutVar("s") > Constant(0)))
```

The formula states that always ("globally", G) in the given interval (`0` to `stopTime`) the value of the observation `s` must be greater than `0`. Such properties can be checked on the output of single simulation runs, yielding true or false per run. These results are then combined in a hypothesis test to determine whether the observed and checked runs provide evidence that a random run from the model will satisfy the property or that it will not. In literature, two main test procedures are described: the *single sampling plan* and the *sequential probability ratio test* (Sen et al. 2005, Legay et al. 2010). Both tests are implemented in SESSL. Whereas the single sampling plan initially determines a number of simulation runs that is guaranteed to yield enough evidence for a decision, the sequential probability ratio test executes batches of simulation runs and after each batch checks the executed runs for evidence. Thus, the sequential probability ratio test usually requires fewer replications to come to a decision. It also is an approximate procedure, although it is argued that in practice it often gives correct results (Legay et al. 2010, p.5).

For a statistical model-checking experiment in SESSL a test procedure must be specified. Both test procedures require four parameters: the probability threshold `p`, the probability bounds for Type I and Type II errors `alpha` and `beta`, and the width of the indifference region `delta`. The indifference region increases the efficiency of the test, assuming the real probability is not too close to the probability threshold. For details on the significance of these parameters see Legay et al. (2010). In general, smaller values for `alpha`, `beta`, or `delta` result in more simulation runs that need to be executed, but also a stronger statement about the model. The following listing shows the configuration of a test procedure in a statistical model-checking experiment in SESSL.

```
test = SequentialProbabilityRatioTest(
  p = 0.8,
  alpha = 0.05,
  beta = 0.05,
  delta = 0.05)
```

The statements `prop = ...` and `test = ...` are provided by the trait `StatisticalModelChecking`. This trait also provides the new hook `withCheckResult`, which can be used similarly to the result processing as described in Section 3.3. Thus, to make the experiment in Figure 2, a statistical model-checking experiment, the trait and the specification of the test procedure, the property, and the result handler must be added. The experiment now verifies that in a random simulation run the prey species does not die out with a probability of at least 80%.

```
execute {
  new Experiment with StatisticalModelChecking /* other traits */ {

    /* experiment setup */

    test = SequentialProbabilityRatioTest(p = 0.8,
      alpha = 0.05,
      beta = 0.05,
      delta = 0.05)

    prop = MITL(G(0, stopTime)(OutVar("s") > Constant(0)))

    withCheckResult { result =>
      println(result.satisfied)
    }
  }
}
```

4 EXTENDING SESSL

So far, we have shown how SESSL can be applied to define and execute simple as well as complex simulation experiments. We now switch from the perspective of the user, who is applying existing features of SESSL, to the perspective of the developer, who is adding new features to SESSL. In fact, with SESSL being an internal domain-specific language, this switch is not that significant, as even the users of SESSL write valid Scala code. As exemplified in Figure 4, Scala's features can also be helpful to structure experiments, not only to add new features. Thus, for a user who is sufficiently proficient in Scala, modifying and extending SESSL is a natural complement to using the predefined features.

4.1 Developing a New Generic Trait

On several occasions, we noted that it is possible to inject custom Scala code into an experiment specification to implement features that are not yet part of SESSL. We now demonstrate how such custom code can be made reusable by wrapping it in a new trait. In general, reusing code is a major aspect of the design philosophy of SESSL, and the code is frequently refactored to avoid code duplication. This leads to a steady growth of well-tested, reliable features, on which the implementations of new features can build. As a side effect, refactoring also reveals salient aspects of defining and executing simulation experiments in general, but also for specific experiment types, which is reflected in the existing bindings and traits.

As an example for implementing a new generic, simulation-system-agnostic trait, we show how a trait to import an experiment design from a CSV file can be realized. Consider, for example, an experiment that shall yield some performance measure of a simulation model under different parameterizations, which are given as an Excel sheet. This new trait will allow for using the list of parameterizations as an input to the experiment, by only converting it to a CSV file. The CSV shall have a header with the parameter names and each further line shall be a design point. Thus, the file might look like this:

```
a,b,c
0,0,0
1,1,1
```

```
0,1,0
1,0,1
```

Our new trait will read the file, convert each line after the header to a model parameterizations, and configure the experiment to run all parameterizations. This is triggered with the new method `designFromCSV` that is implemented in the new trait `CSVInput`. The code wrapped in this method could also be written directly into an experiment specification, but is made easily reusable with this trait.

```
1  trait CSVInput {
2    this: AbstractExperiment =>
3
4    def designFromCSV(file: File): Unit = {
5      import scala.collection.JavaConverters._
6
7      val parser = CSVParser.parse(file,
8                                   Charset.defaultCharset(),
9                                   CSVFormat.DEFAULT.withHeader())
10
11     val headerMap = parser.getHeaderMap.asScala
12
13     val designPoints = for {
14       record <- parser.iterator().asScala
15     } yield {
16       val varValues = headerMap.map {
17         case (varName, idx) =>
18           VarSingleVal(varName, record.get(idx).toDouble)
19       }.toList
20       Config(varValues: _*)
21     }
22
23     configs(designPoints.toList: _*)
24   }
25 }
```

The implementation relies on the Apache Commons CSV library (The Apache Software Foundation 2018), which is written in Java and, thus, compatible with Scala. However, as some conversions between Java and Scala data types are required, the corresponding helper methods provided by Scala are imported (line 5). The library is used in lines 7–9 to parse the file. Subsequently, the header is obtained (line 11) and, using the header, each further line of the CSV file (line 14) is converted to SESSL's internal format for variables (line 18 and 20). This results in a number of instances of the class `Config`, which is SESSL's representation of a model parameterization. Finally, the method `configs` is invoked on the `Config` instances. This method is available here as it is defined in `AbstractExperiment`, which is declared as the self-type of our new trait in line 2. Declaring such a self-type is a common pattern when defining experiment traits. First, it gives access to the methods defined in `AbstractExperiment` and, second, it ensures that the trait can only be mixed in instances of `AbstractExperiment`. As `AbstractExperiment` is the superclass for all concrete `Experiment` classes in simulation system-specific bindings, our new trait and our new method are automatically available for all existing and future bindings. In fact, the new method can be invoked with one line in any experiment after mixing in the new trait:

```
1  execute {
2    new Experiment with CSVInput /* other experiment traits */ {
3
4      /* other experiment setup */
5
6      designFromCSV("design.csv")
7    }
8  }
```

Other sources of experiment designs, such as XML files or SQL databases, can be connected to SESSL in a similar fashion by loading the design and translating it to invocations of methods in SESSL's core.

4.2 Developing a Binding for a Simulation System

The development of a new trait is rather straightforward, as it has few dependencies on existing code. Writing a new binding for a simulation system is typically a bit more involved and requires some knowledge of the SESSL core (see Figure 3). In this section, we shortly sketch the typical steps in developing a new binding for an external simulation system.

Legal issues As a first step, it is important to check under which legal conditions the external software can be integrated. For example, if the external software is licensed under the GNU General Public License (GPL), a direct integration of its API into SESSL is not possible, as SESSL's Apache License is not compatible with the GPL. In such cases, it might still be possible to communicate with the external software via the operating system, for example by invoking a binary from SESSL. Alternatively, the binding could be developed with a loose coupling to SESSL as a stand-alone project. A related question is whether the end user will have to install the external tool separately to use the binding or if the binding can supply the needed binaries itself. The latter is straightforward if the external tool is available as a Maven dependency.

Getting to know the interface The next step is to survey the interface(s) provided by the external tool. If it provides a public API, it can be directly communicated with, especially if it is JVM-compatible. Simulation tools that run as native binaries often provide a command-line interface that can be used to start simulation runs. Regardless of the integration method, it must be decided which features of the external tool shall be available in the binding.

Designing a first prototype A new binding starts by defining the class `Experiment` as a subtype of `AbstractExperiment`, which contains code for the basic features of experiments. By mixing in additional traits from the core, more features can be reused. For example, the trait `DynamicSimulationRuns` is useful for defining a binding where additional simulation runs can be started at runtime. However, the code for communicating with the external tool must be written from scratch. Starting from simple code to start a run with a certain stop condition, the feature set of the binding can be extended stepwise, until all targeted features are included.

Refactoring If not already done, the code produced should now be separated into the main `Experiment` class and the accompanying traits. For example, the code for recording observations from the simulation run goes into the trait `Observation`. By sticking to the naming scheme used by other bindings, SESSL experiments are more easily portable between bindings. This is also a good opportunity to convert the simple experiments, which are typically used to test the binding during development, into unit tests. If they are placed in Maven's default locations for tests, the tests can be conveniently executed in an IDE or the command line. Thus, they can serve as regression tests for future work on the binding.

The amount of time necessary to implement a new binding for an external tool depends on how directly the experiment specification can be mapped to the API of the external tool. The amount of code per binding depends on how much reusable code is available. For example, implementing the binding for the simulation library pSSAlib (Ostrenko et al. 2017) required about two days of work by one developer and about 200 lines of code.

5 CONCLUSION

Two general approaches to define and execute simulation experiments can be distinguished, each with specific trade-offs. First, experiments can be implemented as programs in a general-purpose programming language. Second, experiments can be specified with a tailored tool, for example a graphical user interface. Whereas the former approach is flexible, but requires programming knowledge to create and read experiment specifications, the latter approach is user-friendly, but also constrained to a pre-defined feature set. SESSL aims to close the gap by combining both approaches in an internal DSL. It simultaneously provides the

full expressive power of its host language Scala, but also language constructs that facilitate the definition of simple experiments with minimal syntactical overhead. In that sense, SESSL implements Alan Kay's quote that "simple things should be simple, complex things should be possible" (Leuf and Cunningham 2001, p. 141).

Additionally, we have shown how recurring parts of SESSL experiments can be factored out and made available for reuse. This facilitates a natural growth of the language. When defining an experiment, code for newly implemented experiment aspects can be directly added to the experiment specification. Later on, the implementation can be extracted, put into a new trait, and become part of the language itself. In subsequent experiments, the code can be reused by adding a simple statement to the experiment specification. This way, the trade-off between flexibility and readability of the language can be influenced very easily.

It is possible to consider the Scala parts of experiment specifications as fixed syntax and only use SESSL by locally editing single experiment parts. However, its host language quickly becomes a factor in the practical usage of SESSL. Especially for more complex experiments, a certain level of programming skills is required. For example, the target function of an optimization experiment must be implemented by computing a real number value based on observed simulation trajectories. For such cases, however, user-defined program code is arguably inevitable to realize non-trivial functions. Furthermore, the code required for such functions is rarely complex and, thus, does not require many programming skills. Lastly, coding in Scala is well supported in various IDEs, which we recommend for working with SESSL anyway. Porting the ideas and concepts of SESSL to a different host language, for example Python, could lower the entrance barrier to defining experiments. However, whether this balances out the advantages of Scala's strong type system and syntactical flexibility is questionable.

The potential of Scala has not yet been fully exploited in SESSL. One of the main reasons for this is that SESSL has been carefully designed to be usable by non-programmers as much as possible. For example, observables are currently identified by strings, which allows for a succinct syntax. A language design that is more geared towards programmers could use local variables instead, which could be leveraged to provide type safety and avoid runtime errors. However, this would expose users to Scala keywords and concepts even for simple experiments. We will further explore this trade-off by extending SESSL while maintaining the balance between usability for non-programmers and exploitation of Scala's power.

REFERENCES

Agha, G., and K. Palmskog. 2018. "A Survey of Statistical Model Checking". *ACM Trans. Model. Comput. Simul.* 28(1):6:1–6:39.

Ewald, R., and A. M. Uhrmacher. 2014. "SESSL: A Domain-specific Language for Simulation Experiments". *ACM Trans. Model. Comput. Simul.* 24(2):11:1–11:25.

Helms, T., T. Warnke, C. Maus, and A. M. Uhrmacher. 2017. "Semantics and Efficient Simulation Algorithms of an Expressive Multi-Level Modeling Language". *ACM Trans. Model. Comput. Simul.* 27(2):8:1–8:25.

Legay, A., B. Delahaye, and S. Bensalem. 2010. "Statistical Model Checking: An Overview". In *Runtime Verification*, edited by H. Barringer et al., 122–135. Berlin, Heidelberg: Springer.

Leuf, B., and W. Cunningham. 2001. *The Wiki Way: Quick Collaboration on the Web*. Boston, MA, USA: Addison-Wesley Longman Publishing.

Leye, S., J. Himmelspach, and A. M. Uhrmacher. 2009. "A Discussion on Experimental Model Validation". In *Proceedings of the UKSim 2009: 11th International Conference on Computer Modelling and Simulation*, UKSIM '09, 161–167. Washington, DC, USA: IEEE Computer Society.

Lukasiewycz, M., M. Glaß, F. Reimann, and J. Teich. 2011. "Opt4J – A Modular Framework for Meta-heuristic Optimization". In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, July 12th–13th, Dublin, Ireland, 1723–1730.

Maler, O., and D. Nickovic. 2004. "Monitoring Temporal Properties of Continuous Signals". In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, edited by Y. Lakhnech and S. Yovine, 152–166. Berlin, Heidelberg: Springer.

Maus, C., S. Rybacki, and A. M. Uhrmacher. 2011. "Rule-based Multi-level Modeling of Cell Biological Systems". *BMC Systems Biology* 5(166).

Moles, C. G., P. Mendes, and J. R. Banga. 2003. "Parameter Estimation in Biochemical Pathways: A Comparison of Global Optimization Methods". *Genome Research* 13(11):2467–2474.

North, M. J., N. T. Collier, and J. R. Vos. 2006. "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit". *ACM Trans. Model. Comput. Simul.* 16(1):1–25.

Ostrenko, O., P. Incardona, R. Ramaswamy, L. Brusch, and I. F. Sbalzarini. 2017. "pSSAlib: The Partial-propensity Stochastic Chemical Network Simulator". *PLOS Computational Biology* 13(12):1–15.

Persson, F., and J. Olhager. 2002. "Performance Simulation of Supply Chain Designs". *International Journal of Production Economics* 77(3):231–245.

Sanchez, S. M. 2005. "Work Smarter, Not Harder: Guidelines for Designing Simulation Experiments". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl et al., 69–82. Piscataway, New Jersey: IEEE.

Schruben, L. 1983. "Confidence Interval Estimation Using Standardized Time Series". *Operations Research* 31(6):1090–1108.

Sen, K., M. Viswanathan, and G. Agha. 2005. "On Statistical Model Checking of Stochastic Systems". In *Computer Aided Verification*, edited by K. Etessami and S. K. Rajamani, 266–280. Berlin, Heidelberg: Springer.

Takari 2010. *takari / maven-wrapper*. https://github.com/takari/maven-wrapper, accessed April 26th, 2018.

The Apache Software Foundation 2018. *Apache Commons*. https://commons.apache.org/, accessed April 26th, 2018.

Volterra, V. 1926. "Variazioni e fluttuazioni del numero d'individui in specie animali conviventi". *Mem. R. Accad. Naz. dei Lincei* 2:31–113.

Warnke, T., A. Steiniger, A. M. Uhrmacher, A. Klabunde, and F. Willekens. 2015. "ML3: A Language for Compact Modeling of Linked Lives in Computational Demography". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz et al., 2764–2775. Piscataway, New Jersey: IEEE.

Wilensky, U. 2018. *NetLogo 6.0.3 User Manual: BehaviorSpace Guide*. https://ccl.northwestern.edu/netlogo/docs/behaviorspace.html, accessed April 26th, 2018.

AUTHOR BIOGRAPHIES

TOM WARNKE is a Ph.D. student in the modeling and simulation group at the university of Rostock. His research focuses on the development of domain-specific languages for modeling and simulation. His email address is tom.warnke@uni-rostock.de.

ADELINDE M. UHRMACHER is a professor at the Institute of Computer Science, University of Rostock and head of the modeling and simulation group. She has been on the editorial boards of a variety of journals, including editor in chief of SCS Simulation (2000-2006) and editor in chief of the ACM Transactions of Modeling and Computer Simulation (TOMACS) (since 2013). Her research focuses on the development of modeling and simulation methods and tools, and their application in diverse fields like demography and cell biology. She is member of the ACM Task Force on Data, Software, and Reproducibility in Publication, and initiated artifacts evaluation procedures for ACM TOMACS and ACM SIGSIM PADS. Her email address is adelinde.uhrmacher@uni-rostock.de.