SESSL: A Domain-Specific Language for Simulation Experiments

(Pre-print)

Roland Ewald and Adelinde M. Uhrmacher, University of Rostock

This paper introduces SESSL (*Simulation Experiment Specification via a Scala Layer*), an embedded domain-specific language for simulation experiments. It serves as an additional software layer between users and simulation systems and is implemented in Scala. SESSL supports multiple simulation systems and offers various features, e.g., for experiment design, performance analysis, result reporting, and simulation-based optimization. It supports 'cutting-edge' experiments by allowing to add custom code, enables a reuse of functionality across simulation systems, and improves the reproducibility of simulation experiments.

Additional Key Words and Phrases: Simulation, Experiments, Domain-Specific Language, Scala

ACM Reference Format:
THIS IS A PREPRINT. See `http://dx.doi.org/10.1145/2567895` for the final version, which appeared in ACM Transactions on Modeling and Computer Simulation, 2014, 24(2). ACM Trans. Model. Comput. Simul. V, N, Article A (January YYYY), 25 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The importance of unambiguous, sound, and simple ways to set up reproducible and efficient simulation experiments can hardly be overstated, particularly when considering the various problems that pervade simulation studies [Pawlikowski et al. 2002] and scientific computing in general [Merali 2010; Joppa et al. 2013]. However, supporting users in setting up simulation experiments is difficult. Typically, the simulation system at hand allows to define various experiment parameters, the names of which are not always clear or even used consistently across the system (let alone others). This is particularly true in a research context, where software development moves fast and the resulting prototypes are rarely documented and maintained as thoroughly as proprietary software.

One way to improve the current situation is to standardize experiment descriptions, but several issues hinder the establishment of such standards. Firstly, there is a natural time lag between the demands of 'power users' and standardization efforts. Hence, many users would have to wait until their desired experiment setups are expressible within the standard — or, more likely, they define their experiments for a specific simulation system instead. This, in turn, means less engagement to standardize experiment descriptions in the first place, leading to a vicious cycle that we suspect is one of the main reasons for the lack of a well-established standard to date. Secondly, any sufficiently expressive description language for simulation experiments will be so feature-rich that sophisticated tools are necessary to support users, e.g., custom editors with syntax highlighting. Developing such tools and integrating them into simulation systems requires considerable effort. Thirdly, standardized descriptions alone are no panacea for sound experiments, since much depends on how a description is interpreted by a simulation system. If an experiment description contains insufficient detail (e.g., regarding the usage of random number generators), the simulation system itself has to fill the gaps, so that the results may be irreproducible with other systems.

To address the above issues, we propose to add a separate software layer on top of simulation systems. This layer shall provide a system-independent application programming interface (API) for specifying simulation experiments. Further, we argue that such an API should be realized as an embedded domain-specific language (DSL), to maximize its readability and usability. A DSL is *"a [...] language [...] that offers, through appropriate notations and abstractions, expressive power focused on [...] a particular problem domain."* [van Deursen et al. 2000, p.1]. A DSL is called *embedded* if it is implemented with constructs of a general-purpose programming language, the so-called *host language* of the embedded DSL. In other words, embedded DSLs are APIs that *"[...] should have the feel of a custom language [...]"* [Fowler 2010, p. 28].[1]

Since, technically, an embedded DSL is just an API, it can be mixed with custom code. Thus, even 'power users' can use it for their experiments: they merely need to provide additional code for those peculiarities that are not yet supported. By contributing support for hitherto neglected experiment facets, these users may thus help to 'grow' the language. Extending an embedded DSL is typically straightforward, as it simply means to extend an API written in the host language. The ability to evolve a language bottom-up, driven by demand, seems particularly important in the field of simulation, with its large (and sometimes disconnected) sub-communities. Furthermore, an embedded DSL can be used with the same tools as its host language, so that convenient features like syntax highlighting or code completion are available from the onset, without additional development efforts. Serving as a layer between experimenters and simulation systems, the DSL implementation may also provide common checks regarding the soundness of the experiment (e.g., whether some factors are accidentally defined twice).

Setting up an experiment with an embedded DSL means to *program* its execution, which leaves — in contrast to experiment descriptions — no room for ambiguity. For example, consider an experiment that does not define which random number generator to use. With a program, the effect of this omission is unambiguous (e.g., a default random number generator is used). With an experiment description, on the other hand, this issue has to be resolved by each simulation system that interprets the description. This leaves room for ambiguity: systems may have different defaults (or return different errors), and it might be difficult to investigate what generator had been used in a given experiment. To emphasize this distinction, we speak of experiment *specification* instead of experiment description. While reproducibility is still only ensured when using the same simulation system (as with experiment descriptions), a specification makes the experiment's dependence on certain software artifacts explicit (e.g., via `import` statements) and prevents unintentional misuse (e.g., by provoking compile errors in their absence). Also, default settings can be defined at the level of the DSL and reused across simulation systems, thereby improving the odds for keeping experiments reproducible even across different systems.

To illustrate the above advantages, we present SESSL, an embedded DSL for simulation experiments. It is written in Scala, a programming language compatible with Java, and thus easy to use with many simulation systems. We designed it for maximal simplicity, so that its users do not need to be knowledgeable programmers. At the same time, SESSL allows to set up complex simulation experiments. In Section 2, we explain the design of SESSL and its overall structure. To illustrate its broad applicability, we then discuss several sample

[1]According to [van Deursen et al. 2000], the term 'embedded DSL' was introduced in [Hudak 1996].

experiments (Section 3). Afterwards, we discuss our approach in the context of related work (Section 4).

2. SIMULATION EXPERIMENT SPECIFICATION VIA A SCALA LAYER (SESSL)

In this section, we first detail the requirements that an additional software layer for experiment specification should meet (Section 2.1) and what makes Scala a suitable host language for a DSL realizing such a layer (Section 2.2). Then, we describe SESSL's design (Section 2.3) and its concrete syntax (Section 2.4).

2.1. Requirements

First and foremost, the language should be easy to learn. Experiments should be simple to understand, even for a cursory reader. At the same time, the language should be expressive enough to cover most usage scenarios that arise in practice, and its notation should be concise.

To optimally assist users, the language should make it easy to define meaningful experiments and hard to define inconsistent ones. For example, its implementation should warn a user who configured a data sink for simulation output although the experiment does not generate any such output. The language should also be largely independent of a specific simulation system. This allows users to use the same (or very similar) notation to work with different systems. Experiment setups would become portable, which helps to cross-validate results. Experimenters should be able to reuse (parts of) experiment specifications, also from within other software systems.

To gain broader acceptance, it should be straightforward to add support for another simulation system. At first sight, this problem seems easy to solve, because an embedded DSL hides the concrete API of a simulation system. On the other hand, a simulation system may offer some specialized experimentation techniques not yet accessible via a standardized interface, e.g., a particular technique for sensitivity analysis or a new way to store simulation data. If the specifics of a simulation system were *completely* hidden and developers would have no way to add system-specific constructs, these facets could not be configured easily. As pointed out in the introduction, this discourages 'power users' from adopting experimentation standards. Therefore, the language itself needs to be easy to customize and extend.

Allowing developers of simulation systems to define system-specific features helps to evolve the overall language. If, at some point, the way to specify a certain experiment facet becomes accepted within the community, incorporating it into the 'core' of the language should be easy. In any case, experimenters should typically not be bothered by this,[2] i.e., the combination of system-specific and general language constructs needs to be seamless and unobtrusive.

2.2. Why Scala?

We chose the general-purpose language Scala [Odersky et al. 2011] as a host language for SESSL because of two reasons: firstly, it is fully interoperable with Java, a popular language among simulation system developers. This makes it particularly easy to support simulation systems executed on the Java virtual machine (JVM). Of course, SESSL can also work with other kinds of simulation systems (as we demonstrate in Section 3.3.2), but this requires more development effort. Secondly, Scala has many features that make it an ideal choice for developing embedded domain-specific languages (e.g., see [Odersky et al. 2011, p. 727 et sqq.]). In the following, we focus on those Scala features that are essential for the design of SESSL.

[2]Unless they aim at making their experiments reproducible across different systems.

Scala is statically typed and combines object-oriented and functional programming. Functions are regarded as objects and can thus be passed to other functions. Functions can have arbitrary names, such as '`+`' or '`*`', and can be used in infix notation, so that they look like built-in operators. Functions can have default parameter values and can be invoked with named parameters. For example, a function defined by "`def f(x:Int=1,y:Int=2,z:Int=3) = x+y+z`" can be called with `f(z=5)`, which yields `8`. Another feature that makes Scala well-suited for DSL creation is that function calls with a single argument can be written with curly braces, so that they look like keywords in other languages (e.g., `try{...}` in Java, see [Odersky et al. 2011, p. 173 et sqq.]). This enables structuring (and nesting) DSL elements in a way with which many users are already comfortable.

To further improve readability, Scala allows to define so-called *case classes* [Odersky et al. 2011, p. 270]. Among other things, case classes can be instantiated without a `new` keyword. Since constructors can also have named parameters with default values, a case class defined by "`case class MyType(x:Int=0,y:Int=0)`" can be instantiated by simply writing `MyType(y=1)`.

Scala supports automatic type conversions via *implicit views*. Whenever an object's type does not provide an invoked method, a special function within the current scope may generate an object of a type that does provide this method. For example, invoking `"a".f` results in a compile error, since objects of type `String` do not have a method `f`. However, if the compiler can find an implicit conversion function `g` in the current scope that is able to convert a `String` object to an object of a type that *does* have such a method, this gets essentially rewritten to `g("a").f`.[3]

Scala enhances Java's notion of interfaces by offering *traits*. Traits are interfaces that may also contain method definitions and member variables, and therefore can implement own functionality. A trait is said to be *abstract* if it has at least one method that is merely declared, i.e., its implementation has to be provided by a subtype. Traits cannot be instantiated, but arbitrarily many traits can be *mixed* into a class. This so-called mix-in composition is a powerful feature, because the traits mixed into a class may alter its functionality. By declaring a so-called *self-type*, a trait can restrict the types into which it can be mixed, thereby also gaining access to all methods of this type. Furthermore, traits can also override methods of the class they are mixed into, and by doing so their functionality can be 'stacked' [Odersky et al. 2011, p. 226].

2.3. Language Design

To ensure simplicity and readability, we propose to explicitly compose experiment specifications from 'sub-specifications' for particular experiment facets (Section 2.3.1). To ensure independence of simulation system specifics, implementations of such sub-specifications need to be split up (Section 2.3.2). To ensure that identical techniques realized by different systems can nevertheless be identified as such, they should be associated via Scala's type system (Section 2.3.3). Finally, to ensure that experiment results are easy to access, SESSL offers a common, simulator-independent, and extensible result handling mechanism (Section 2.3.4).

2.3.1. Experiment Composition. Besides essential settings, e.g., which model to simulate, there are many non-essential experiment facets that are merely important in some situations, e.g., how to plot simulation output. Experimenters should only have to deal with those facets if they are relevant for a given experiment. This reduces the complexity of the specification task, as only a subset of all language constructs has to be available. Therefore, SESSL specifications are composed from *experiment facets*.

[3]Java programmers may think of this as a customizable form of auto-boxing.

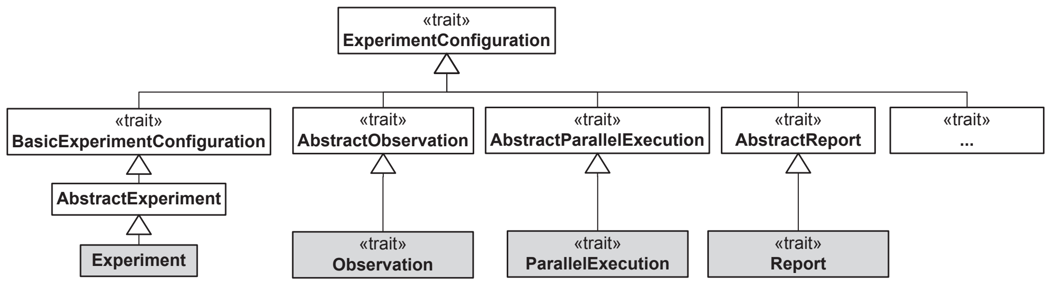

Fig. 1. The hierarchy of central types in the SESSL core (white) and their relation to the types provided by a binding (gray), which implement their functionality for a concrete simulation system.

Experiment facets are realized as self-typed traits that get mixed into an `Experiment` class, which contains the essential settings. Self-types allow to express dependencies between experiment facets. For example, specifying how recorded data shall be stored implies that data is recorded in the first place. Thus, a trait for configuring a data sink can express, via its self-type, that it may only be mixed into experiments that record data. Trying to do otherwise will result in a compile error, so that experimenters notice the problem immediately.

Before a SESSL experiment starts, a configuration method is called for each experiment facet, i.e., trait, that has been mixed in. This way of component composition is also called the *cake pattern*, as each trait adds an additional layer of functionality to the original class. The cake pattern is well established and used for the Scala compiler itself (e.g., see [Odersky and Zenger 2005]). Using the cake pattern in SESSL leads to a modular 'interpreter' [Hudak 1996] that consists of a component for each of the different experiment facets: ideally, each trait provides both the methods to configure a certain experiment facet *and* the interpretation logic to map this configuration to the given simulation system, e.g., by calling its API. Another advantage of the cake pattern is its extensibility. Developers can easily add custom traits for experiment facets not yet supported (an important requirement, see Section 2.1).

2.3.2. Independence of Simulation Systems. While SESSL serves as an additional layer of abstraction, the specifics of a simulation system must be taken into account within the interpretation logic of the DSL, i.e., the code that configures a simulation system according to the given experiment specification. This code is, in general, relatively easy to write for embedded DSLs [Zdun and Strembeck 2009, p. 29]. However, as integrating additional simulation systems should be easy (see Section 2.1), programmers of interpretation logic need to be supported as much as possible. We therefore split the type hierarchy of SESSL into general components and system-specific components. We refer to the former as the SESSL *core* and to the latter as SESSL *bindings*, one binding for each system to be integrated. In the terminology of [Fowler 2010], the SESSL core provides a 'semantic model' of simulation experiments on which the bindings operate.

The class diagram in Figure 1 illustrates this division: types of the SESSL core are reused via inheritance by the types that realize a binding. All types of the SESSL core that are intended to be used this way have the prefix `Abstract`. They are usually realized as abstract traits. To integrate a simulation system, a developer defines sub-traits that inherit from these abstract traits. This means to implement the system-specific methods declared (but not defined) within the abstract traits of the SESSL core. An abstract trait represents an experiment facet. By developing sub-types for only some of these traits, developers implicitly describe which functionality of a simulation system can be configured via SESSL.

2.3.3. Representing Algorithms. Simulation experiments typically involve the execution of multiple distinct algorithms, e.g., event queues or random number generators. While algorithms may be chosen automatically by the simulation system (e.g., see [Ewald 2011]), there are many situations where experimenters need to exert more control. For example, when the performance of a simulation algorithm shall be investigated, the algorithm and its parameters need to be configurable. Otherwise, stochastic SESSL experiments were — strictly speaking — not even reproducible: if there is no way to set up a specific random number generator (and its seed), the results of two subsequent executions may not be identical. This restriction would exclude a whole range of relevant use cases, e.g., testing (see Section 3.1), and thus needs to be avoided.

Besides parameters, algorithms may also rely on sub-algorithms, which should be fully configurable as well. Thus, SESSL has to provide a simple way of configuring algorithm hierarchies (e.g., see [Himmelspach and Uhrmacher 2007]). Two conflicting issues further complicate algorithm configuration. On the one hand, algorithm implementations depend on the specific simulation system that provides them. The set of available algorithms may change over time, and it also varies from system to system. All this would call for a system-dependent definition of algorithms. On the other hand, however, many implementations are interrelated in some sense, e.g., they may be able to simulate the same kind of model or realize the same fundamental approach. Making these interrelations explicit is important to guide experimenters, e.g., when switching from one simulation system to another.

To resolve this problem, SESSL combines system-dependent case classes with a system-independent type hierarchy of marker traits. A marker trait is a trait that does not declare any methods and is thus merely used for ‘marking’ a type regarding certain properties. The long term goal of such marker traits is to represent knowledge on simulation concepts and their relations. This knowledge should be shared among all bindings, very much in the spirit of an ontology. Developers of SESSL bindings have to define a class (typically a one-liner) for each specific algorithm implementation that shall be explicitly configurable by experimenters. The marker traits mixed into such a class define how the algorithm implementation relates to general concepts and methods for simulation, e.g., whether it is an approximative method or what kind of problem it solves. By providing a class with certain marker traits, a binding declares both properties and application domain of a simulation (sub-)algorithm implementation. This helps experimenters to find related approaches via Scala’s type system. In future, additional traits could easily associate these classes with corresponding (or related) concepts in existing ontologies for modeling and simulation, such as DeMO [Miller et al. 2004] or KiSAO [Courtot et al. 2011]. The potential benefits of representing ontologies via type systems has already been noted by others, e.g., see [Despeyroux 2008].

One perceived drawback of this solution could be that experimenters have to explicitly distinguish between, for example, cellular automata simulators from two different simulation systems. While both classes would refer to the same kind of algorithm, as indicated by their (marker) traits, they would still refer to *different* implementations of said algorithm and these may indeed have *different* characteristics (e.g., w.r.t. runtime performance). Therefore, being able to distinguishing both implementations is necessary to ensure reproducibility.

2.3.4. Result Handling. Result handling is a challenging issue in simulation experiments, because there are various kinds of result. The type of result that first comes to mind are metrics based on the model state as simulation time evolves. Such results can be represented as trajectories or time-series (depending on the kind of simulation). For many stochastic simulations, however, only statistics that aggregate these fine-grained results are of interest: for example, the mean value of a variable at a given point in simulation time, averaged over all replicated trajectories.

There are also many other kinds of results, depending on the purpose of a simulation experiment. For example, experiments to analyze simulator performance rarely collect sim-

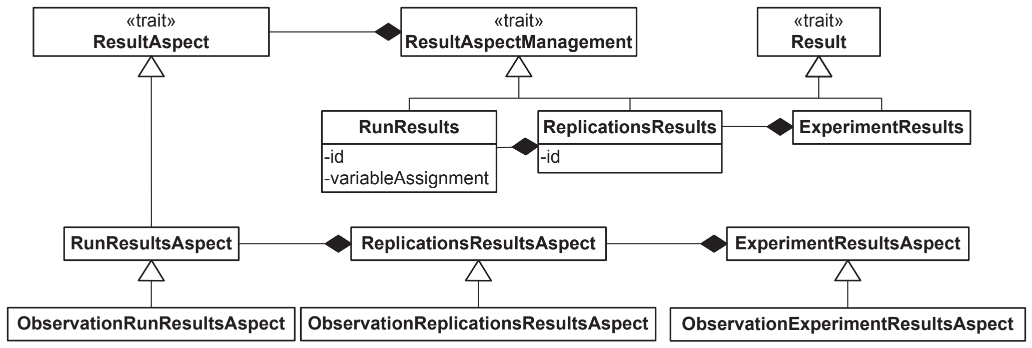

Fig. 2. The type hierarchy of results and result aspects. `RunResults` and `ReplicationsResults` have a unique identifier to tell them apart (these may be system-specific). `RunResults` also stores the assignment of model variables that was used to generate the results. This allows to group all runs with identical assignments to `ReplicationsResults`.

ulation output data, which would only slow down the evaluation. Instead, they collect performance metrics, e.g., run times. As SESSL shall be easy to use and extensible, we have to deal with two issues arising from this. Firstly, experimenters should be able to specify *which* kind of result shall be processed on *what* level of aggregation. Secondly, developers should be able to add support for new kinds of results later on, so that the language can evolve (see Section 2.1).

In terms of aggregation, it seems most useful to distinguish between the results of single runs, the results of simulation replications (i.e., repeated runs with the same parameter values), and the overall results of an experiment (e.g., see [Himmelspach et al. 2010]). SESSL allows experimenters to process results on each of these aggregation levels, by following the DSL pattern *nested closure* [Fowler 2010, p. 403]. Experimenters can specify (arbitrarily many) functions for post-processing simulation results. The functions will be invoked during experiment execution and, as they are closures, may refer to variables in their scope (e.g., to write a result into another data structure for later analysis).

The second issue, i.e., allowing developers to specify new kinds of results, is solved by distinguishing between results and result *aspects*. A run, a set of replications, or a whole experiment can each be associated with any number of result aspects. We use 'aspect' as a non-technical term here, not to be confused with the term 'aspect' from aspect-oriented programming, which refers to functionality for a cross-cutting concern (e.g., error handling). In contrast, result aspects refer to different kinds of data generated by simulation experiments.

Figure 2 shows the type hierarchy to represent results in SESSL. It comprises two subhierarchies, one for results (upper right corner) and one for result aspects. The result types are derived from the type `Result` and have aggregation relationships among each other: for example, an object of type `ReplicationsResults` contains (i.e., references) all results of the individual runs it represents. Each result also inherits from `ResultAspectManagement`, which provides the functionality to manage the data of additional result aspects. Result aspects form a similar type hierarchy: they are all derived from `ResultAspect` and again have aggregation relationships among each other. This makes it easier to develop aggregated result aspects, which can access results of their own aspect from lower aggregation levels.

Our approach to result handling enables experimenters to express exactly *which* result aspect to process (e.g., only execution times), and on *what* aggregation level. Experiment facets that offer new kinds of results simply provide additional methods for result processing (see Section 2.3.1). For example, the trait `AbstractObservation` (see Figure 1) offers methods for working with the simulation outputs of a single run, a set of replications, or a whole experiment. Likewise, the trait `AbstractPerformanceObservation` provides meth-

```
1  import sessl._
2  import sessl.james._
3
4  execute {
5    new Experiment with Observation {
6      model = "file-sr:/./SimpleModel.sr"
7      scan("r1" <~ (0.5, 1, 1.5))
8      replications = 10
9      stopCondition = AfterWallClockTime(seconds=1) and AfterSimTime(10e4)
10     observe("A")
11     observeAt(range(100, 50, 9000))
12     withRunResult {
13       result => println(result ~ "A")
14     }
15   }
16 }
```

Fig. 3. A simple SESSL experiment using JAMES II. Scala keywords are printed in blue.

ods for working with the performance data of a single run, a set of replications, or a whole experiment.

The management and processing of results is implemented in the SESSL core, and hence can be reused across simulation systems. By adding new types of result aspects and additional methods to handle them, developers can easily extend the current result processing capabilities across different aggregation levels.

2.4. Syntax & Features

2.4.1. Introductory Example. We introduce the syntax with a small sample experiment, shown in Figure 3. The SESSL core is located in package `sessl`, with bindings located in sub-packages named after the software systems. Thus, an experimenter imports the SESSL core (line 1) and the binding for the simulation system to be used, in this case the open source modeling and simulation framework JAMES II [Himmelspach and Uhrmacher 2007] (l. 2). Then, the experimenter executes a simulation experiment by calling the function `execute` (l. 4–16) from the SESSL core with a new instance of an (anonymous) sub-class of type `Experiment` (l. 5–15). The experiment involves observing simulation output, which is declared by mixing in the trait `Observation` (l. 5). Both `Observation` and `Experiment` are provided by the binding for JAMES II, but inherit from abstract types of the SESSL core (see Figure 1).

Scala class constructors are written directly into the body of the class. They are used in SESSL to define the details of the experiment (l. 6–14). At first, we specify the model file to be used (l. 6). While this line looks like an assignment, it invokes a function to set the model location. This allows to internally perform additional checks (e.g., does the file exist?), which may also be system-specific (by overriding it, see Section 2.4.3). SESSL relies on this syntactic simplification offered by Scala for most ‘assignments’. Note that there are no restrictions regarding the model format. The specified string will be handed over to the binding and hence can be system-specific. Such dependencies can be avoided by using standardized model formats like the systems biology markup language (SBML) [Hucka, M. et al. 2003] or the Petri Net markup language (PNML) [Weber and Kindler 2003].

Line 7 sets up a scan for model parameter `r1` $\in \{0.5, 1, 1.5\}$. SESSL provides an implicit view for the `String` literal `"r1"`, for which a function named $<\sim$ is defined (see Section 2.2).[4]

[4]This technique is also known as *literal extension* [Fowler 2010, p. 481 et sqq.].

```
 1 import sessl._
 2 execute({
 3   import sessl.sbmlsim._
 4   new Experiment {
 5     model = "./sbmlModel.xml"
 6     stopTime = .01 //[...]
 7   }
 8 }, {
 9   import sessl.james._
10   new Experiment {
11     model = "./sbmlModel.xml"
12     stopTime = .01 //[...]
13   }
14 })
```

Fig. 4. This code fragment executes the same simulation experiment with two different simulation systems, SBMLsimulator (see Section 3.3.1) and JAMES II. The duplicate specifications (of model and stop time) can be avoided by defining a trait and using mix-in composition (see Section 3.1).

The function takes a list of values as an argument and returns a SESSL object that contains both the name of the parameter and the values that shall be assigned to it. This makes constructs like `scan("r1" <~ (0.5, 1, 1.5))` possible, where `scan` is another function provided by the SESSL core, to set up parameter scans.

Line 8 specifies that ten replications shall be executed per model parameterization. Each simulation run is only stopped after at least one second of wall-clock time has passed *and* the simulation time of 10^4 time units has been reached (l. 9). `AfterWallClockTime` and `AfterSimTime` are case classes (see Section 2.2). The function '`and`' between them aggregates both criteria to an internal object representing a conjunction (an '`or`' function is available too). This syntax facilitates the definition of arbitrarily nested stopping criteria. Also note that the instantiation of `AfterWallClockTime` makes use of default and named parameters: e.g., `hours` has default value `0` and is omitted (see Section 2.2).

The amount of a chemical species A within the model shall be observed (l. 10) during the simulation time interval $[100, 9000]$ at equidistant time points (50 time units apart, l. 11). The functions used in these two lines are provided by the mixed-in trait `Observation` (l. 5, see Section 2.3.1). Using, for example, `observe` in an experiment without mixing in `Observation` would cause a compile error.

After each run, the trajectory for A is printed to the standard output (l. 12–14). The event handling code in line 13 defines a function that takes the `result` of a single run as an argument. It is passed to the result handling function `withRunResult` (see Section 2.3.4), which stores it for later invocation. The expression `result ~ "A"` (l. 13) calls the method ~ of object `result` with the argument `"A"`. This returns the trajectory observed for A.

2.4.2. Switching between Simulation Systems. To minimize confusion when switching to another simulation system, concrete types should always have the same name as the SESSL type they inherit from, but without the prefix '`Abstract`'. This means in case a common format is used for model specification, e.g., SBML [Hucka, M. et al. 2003], one can 'port' an experiment specification from one system to another by merely replacing the import statement for the binding. This is illustrated in Figure 4 (see `import sessl.sbmlsim._`, l. 3, and `import sessl.james._`, l. 9).

Switching between different interpretations of Scala code by package names is a widely adopted technique. For example, it is also used in the Scala collections API, where packages named `mutable` and `immutable` contain types of the same name.

2.4.3. Extending SESSL. The only type that must be provided by a binding is a sub-class of `AbstractExperiment`, which should be named `Experiment` (see Figure 1). Additional experiment facets are supported by inheriting from the corresponding abstract types in the SESSL core. We selected the currently supported experiment facets (e.g., for observation, parallel execution, or performance analysis) based on features provided by the experimentation layer of JAMES II, which already offers a powerful programming interface to set up experiments [Himmelspach et al. 2008; Ewald et al. 2010].

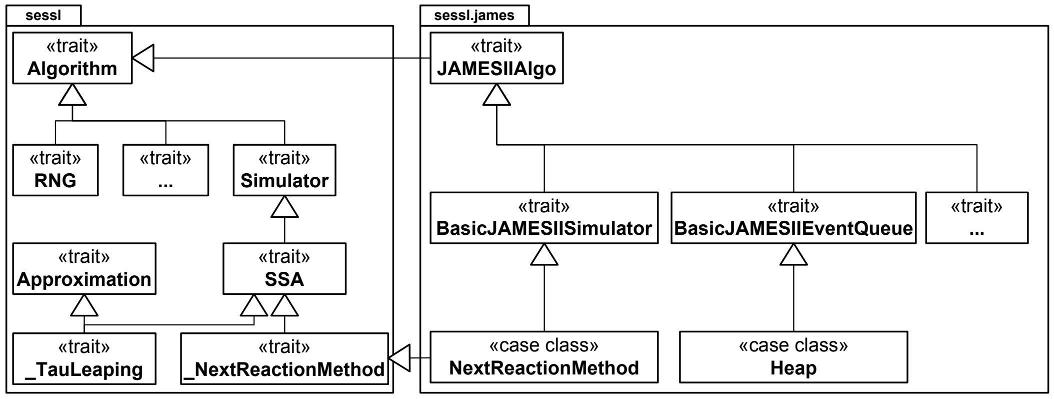

Fig. 5. Super types of `sessl.james.NextReactionMethod` and their relations. Cross-package inheritance relations of several other types (e.g., `Heap`) are omitted for clarity.

Figure 5 shows how new classes can be added to represent concrete algorithm implementations (see Section 2.3.3). The marker traits defined in package `sessl` are system-independent and should be used to categorize available implementations. Marker traits that refer to specific algorithms are prefixed with '_', to distinguish them from actual implementations. If a default implementation of a well-known algorithm is provided, one should therefore just omit the prefix '_' in its name, as done with `sessl.james.NextReactionMethod` (see Figure 5). This makes the algorithm representation easy to find, e.g., by the code completion mechanism of an integrated development environment (IDE).

The other types defined by the binding, i.e., in package `sessl.james` (see Figure 5), shall make the representation of additional implementations as simple as possible. Since JAMES II itself distinguishes between plug-in types and plug-ins (see [Himmelspach and Uhrmacher 2007]), this is reflected by an additional layer of sub-types (e.g., `BasicJamesIISimulator`), one for each plug-in type that needs to be configurable by the user. Each case class (e.g., `NextReactionMethod`) represents a plug-in and hence merely points to the corresponding JAMES II factory. Since Scala's notation for new types is concise — most types in Figure 5 are defined with a single line of code — it can be easily used to represent additional knowledge on simulation approaches. Bindings for other complex simulation systems can follow a similar approach. Note that the intricacies of the algorithm type hierarchy are mostly transparent to experimenters, while it still allows to check the correctness of a configuration. For example, an experimenter who tries to use a simulation algorithm as a random number generator provokes a compile error and is thus notified immediately.

Bindings may also offer shorthand notations to conveniently access results, such as the method $\sim$ to access a variable's trajectory (see Figure 3, l. 13). To support this, SESSL provides some auxiliary traits that can be mixed into new result aspect types, so that standard mathematical functions (e.g., min, max) can be reused, as well as a simple way to select result sub-sets. For example, the construct `results.having("r1" <~ 0.7).max("A")` selects the maximal (last-observed) value of A for all simulation runs where the model variable `r1` has been set to 0.7, i.e., this simplifies the aggregation over result sub-sets (see Supplementary Material, Section C).

3. SAMPLE EXPERIMENTS

3.1. Testing Simulation Software

Consider a scenario where a new auxiliary data structure is developed for a simulation algorithm. For example, the default event queue implementation shall be replaced by a heap. Besides employing unit tests to check whether individual functions of the new heap work, integration tests should check whether it also works in its execution context. One possibility is to let the simulation algorithm run with both its default data structure and the new heap implementation, and to compare their results afterwards. However, if setting up suitable simulation experiments is too difficult, developers may neglect integration testing altogether and thereby increase the risk of undiscovered bugs. The larger and more complex a simulation system becomes, the more simulation experiments are required for integration testing, and the more effort is required to specify and maintain them. Using SESSL is beneficial in this context, as its experiment specifications are rather short, concise, and reusable across simulation systems. This allows, for example, to define a 'test library' of useful simulation experiments for integration testing. The relevant experiments would just need to be 'instantiated' for the simulation system under testing, saving its developers much programming effort.

Figure 6 shows some code to illustrate this way of using SESSL. At first (l. 4–16), a new trait `SomeTestSetup` is specified, which uses a self-type (l. 5) to declare which (abstract) experiment facets it relies on (see Figure 1). It then configures the experiment specifics (l. 6–15) in the same way as seen before in Figure 3. The only new specification elements refer to result reporting (provided by trait `AbstractReport`, see Figure 1), which allows to build a report document from experiment results (l. 12–14). Here, a new section titled *Results* is declared, which contains a histogram of the (last) observed amounts of species `S3` for each of the 200 replications. The user may also specify axis labels and a title. How the report is generated depends on the simulation system at hand. JAMES II currently uses LaTeX, R, and SWEAVE [Leisch 2002].[5]

To (re-)use this pre-specified experiment, a developer simply creates a custom class that extends `Experiment` and mixes in `SomeTestSetup` (as well as all concrete traits it requires, provided by the binding). In this example, `ParallelExecution` is additionally mixed in to speed up experiment execution (l. 22, see Section 3.3.1), and a constructor is defined that specifies to use a single simulator (l. 24) and to name the result report accordingly (l. 25). The rest of the sample code (l. 28–33) illustrates how experiments for two configurations of the Next Reaction Method (a stochastic simulation algorithm for chemical reaction networks [Gibson and Bruck 2000]) — with and without our hypothetical new heap implementation — can be instantiated (l. 28–29), executed (l. 32), and evaluated (l. 33).

The approach sketched out in Figure 6 allows to reuse testing and evaluation code across simulation systems. In other words, one separates the concern of experiment specification from the concern of experiment execution. Such a separation implies that changes in the API of a simulation system only need to be addressed in its SESSL binding, but not in the experiment specifications as such. This reduces the effort of maintaining extensive test batteries.

```
//Independent of a simulation system (e.g. provided by a test library):
import sessl._

trait SomeTestSetup {
  this: AbstractExperiment with AbstractObservation with AbstractReport =>
  model = "./some.model.file"
  replications = 200
  stopTime = 1.5
  observe("S3")
  observeAt(1.4)
  withExperimentResult { result =>
    reportSection("Results") {
      histogram(result("S3"))(title = "Species #3 after 1.4 s")
    }
  }
}

//Dependent on concrete simulation system:
import sessl.james._

class JTestSetup(simulatorUnderTest: Simulator)
  extends Experiment with Observation with ParallelExecution
  with Report with SomeTestSetup {
  simulator = simulatorUnderTest
  reportName = "Results of " + simulator
}

val expDefaultSetup = new JTestSetup(NextReactionMethod())
val expHeapSetup = new JTestSetup(NextReactionMethod(eventQueue = Heap()))

//Independent of a simulation system (e.g. provided by a test library):
execute(expDefaultSetup, expHeapSetup)
someStatisticalTest(expDefaultSetup.results, expHeapSetup.results)
```

Fig. 6. A generic simulation experiment specification and how it can be reused for integration tests.

3.2. Runtime Performance Analysis

Another application scenario for SESSL is the runtime performance analysis of simulators. Such analyses are often carried out by the developers themselves, e.g., to evaluate the runtime performance of a newly implemented algorithm. Obtaining performance data on a large scale is also crucial for realizing automatic simulator selection and configuration [Ewald 2011].

The experiment in Figure 7 relies on two additional experiment facets, `PerformanceObservation` and `Report` (l. 2), but `Observation` is not mixed in (i.e., no simulation output will be recorded). After specifying the model, a fixed stop time, and the number of replications (l. 3–5), the representation of algorithms as case classes (see Section 2.3.3) is used to declare two distinct sets of simulation configurations, `tlSetups` and `nrSetups` (l. 7–9). Here, `tlSetups` contains four configurations of τ-Leaping (an approximative simulation algorithm for chemical reaction networks [Cao et al. 2006]) with its parameter $\epsilon \in \{0.02, 0.03, 0.04, 0.05\}$, and `nrSetups` contains four configurations of the Next Reaction Method with different event queues. Assigning the simulator sets to

[5]Some sample plots are shown in the Supplementary Material, Section A.

```
 1  execute {
 2    new Experiment with PerformanceObservation with Report {
 3      model = "java://examples.sr.LinearChainSystem"
 4      stopTime = 1.5
 5      replications = 20
 6
 7      val tlSetups = TauLeaping() scan ("epsilon"<~range(0.02,0.01,0.05))
 8      val nrSetups = NextReactionMethod() scan {
 9       "eventQueue"<~(BucketQueue(),LinkedList(),Heap(),SortedList())}
10
11      simulators <~ (nrSetups ++ tlSetups)
12      executionMode = AllSimulators
13
14      reportName = "Sample Performance Report"
15      withExperimentPerformance { r =>
16        reportSection("Results") {
17          boxPlot(r.runtimesForAll)("Run times for all setups")
18          boxPlot(("TL", r.runtimes(tlSetups)), ("NRM", r.runtimes(nrSetups)))(
                title="Run time comparison for algorithm families")
19        }
20      }
21    }
22  }
```

Fig. 7. A performance analysis experiment for JAMES II.

additional user-specified values facilitates a latter distinction during report creation (l. 18), but is not required. Line 11 joins both sets and specifies them as the `simulators` to be used. At this point, the experiment specification is slightly ambiguous: should *each* of the specified simulators execute 20 replications, or should there be 20 replications *overall*, with the simulation system being able to pick freely among the alternative algorithms stored in `simulators`? Since erring on the second side does *not* result in overly long simulation experiments, this is the default SESSL behavior, i.e., the given replication condition (in this case, a fixed number) has to be satisfied just *once*, and the simulation system may choose any setup from `simulators` to simulate the model. In case the experimenter intends the other interpretation, in which *all* simulators are applied to the given experiment specification one after another, each having to fulfill the replication conditions, this can be specified by setting the `executionMode` to `AllSimulators` (l. 12). In this example, it means that each of the eight simulators executes the model 20 times. The performance measurements are included in a result report, which is done by calling the `withExperimentPerformance` function (l. 15–20), provided by `PerformanceObservation`. In line 16, a single report section named *Results* is created. It contains a box plot for the run times of all individual setups, and a second box plot in which the run times of the τ-Leaping setups are compared to the run times of the Next Reaction Method setups (see Supplementary Material, Section A).

Only execution times are considered so far, but this could be easily extended to other runtime performance aspects (e.g., memory consumption). Moreover, other performance metrics may call for entirely new experiment facets, as they require additional configuration (e.g., processor utility or event throughput).

3.3. Integration of Simulation Tools

So far, all sample experiments relied on the SESSL binding for JAMES II. To show how other simulators can be integrated, we developed SESSL bindings for version 1.0 of SBML-

```
1  import sessl._
2  import sessl.sbmlsim._
3
4  execute {
5    new Experiment with ParallelExecution with Observation
6    with sessl.james.Report {
7      model = "./BIOMD0000000002.xml"
8      set("kr_0" <~ 8042)
9      scan("kf_2" <~ range(30000, 1000, 34000), "kr_2" <~ (650, 750))
10     stopTime = .01
11     observe("x" ~ "ILL", "y" ~ "DLL")
12     observeAt(range(0, 1e-04, 1e-02))
13     simulator = DormandPrince54(stepSize = 1e-06)
14     reportName = "SBMLsimulator Report"
15     withRunResult { result =>
16         reportSection("Run Number " + result.id) {
17           linePlot(result ~ "x", result ~ "y")(title = "Integration Results")
18         }
19     }
20   }
21 }
```

Fig. 8. A SESSL experiment for SBMLsimulator.

simulator [Dräger et al. 2012] and version 4.2.2 of OMNeT++ [Varga 2001]. To show how SESSL can serve as a unified interface for other tasks, such as simulation-based optimization, we integrated the Opt4J framework for meta-heuristic optimization [Lukasiewycz et al. 2011]. To show how specific methods provided by libraries can be reused, we integrated some functionality of the SSJ library for stochastic simulation [L'Ecuyer et al. 2002].

3.3.1. SBMLsimulator. SBMLsimulator is a Java-based software that offers several numerical integrators to simulate SBML models. Figure 8 shows a SESSL experiment specification for SBMLsimulator, defined on a model from the Biomodels database [Li et al. 2010]. The trait `ParallelExecution` (l. 5) specifies that the underlying simulation system may exploit parallelism to speed up the simulation experiment. Note that the experimenter neither specifies *what* shall be parallelized (e.g., simulation runs, output analysis, result storage), nor which particular parallelization approach to use. These details have to be implemented by the simulation system at hand, but do not need to be exposed to experimenters.[6] However, experimenters may still specify how many of the available resources shall be used.

The function `set(...)` (l. 8) can be used to set certain model parameters to a fixed value throughout the whole experiment. Then, a parameter scan over two model parameters, `kf_2` and `kr_2`, is defined (l. 9) and the observation of species `ILL` and `DLL`, accessible as `x` and `y` in SESSL, is configured (l. 11–12). The function $\sim$ (l. 11) links model-specific names, in this case `ILL` and `DLL`, to unique SESSL-specific names. This improves readability and re-usability of specification fragments, as the SESSL names are used throughout the experiment specification, e.g., to access experiment results (l. 17). Line 13 configures the numerical integrator to be used. The last lines (l. 14–19) are concerned with result reporting: for each run, a new report section for this run number is created, as well as a line plot to display the trajectories of `x` and `y`.

[6]The version of SBMLsimulator we used does not support parallel execution, so this is realized by our binding, see Section 3.3.4.

```
 1 import sessl._
 2 import sessl.omnetpp._
 3 
 4 execute {
 5   new Experiment with Observation with EventLogRecording {
 6     model = ("cqn.exe" -> "ClosedQueueingNetA")
 7     set("*.numTandems" <~ 2, "*.numQueuesPerTandem" <~ 3)
 8     replications = 2
 9     stopCondition = AfterSimTime(hours = 10) or AfterWallClockTime(seconds = 10)
10     warmup = Duration(seconds = 20)
11     observeAt(range(1000, 100, 30000))
12     observe("**.queueLength")
13     scan("*.queue[*].numInitialJobs" <~ (2, 4),
14          "*.sDelay" <~ range("%ds", 2, 2, 8) and
15          "*.qDelay" <~ range("%ds", 2, 2, 8) and
16          "*.queue[*].serviceTime" <~ range("exponential(%ds)", 2, 2, 8))
17   }
18 }
```

Fig. 9. A SESSL experiment for OMNeT++.

Note that this specification also relies on JAMES II for result reporting; its `Report` trait is mixed into the SBMLsimulator experiment (l. 6). This is necessary because SBMLsimulator does currently not provide a custom reporting mechanism, i.e., there is no trait `sessl.sbmlsim.Report` to be mixed in, so the experiment specification would not compile (unless lines 14 and 16–18 are removed). In JAMES II, report generation is separated from the experimentation layer and can be used to report data from any source, including any other SESSL experiment. The functions provided by the report trait of the JAMES II binding work on data types defined in the SESSL core, which allows to easily and safely reuse functionality across different simulation systems. This example shows how a *single* SESSL experiment makes two of these systems interoperable — without requiring the experimenter to learn a new API and without any changes to the simulation systems themselves.

3.3.2. OMNeT++. In contrast to SBMLsimulator, the simulation framework OMNeT++ is written in C++, primarily targeted at network simulations, and already offers a powerful *external* DSL for experimentation. External DSLs are not implemented in a programming language, i.e., unlike embedded DSLs they do not rely on the syntax of a host language and therefore require a custom parser. Nevertheless, one can use SESSL to control OMNeT++ simulation experiments. As OMNeT++ does not provide a Java interface to trigger simulation experiments,[7] the binding generates an `omnetpp.ini` file that specifies the experiment to be conducted, runs OMNeT++ as an external process, and reads back the results from the output files after each simulation run.

A sample SESSL experiment for OMNeT++ is shown in Figure 9. It illustrates that experiments for simulation tools as conceptually and technologically different as, for example, JAMES II and OMNeT++, can still be specified in the same way. Furthermore, some specifics of OMNeT++ and its custom experimentation DSL can be easily accommodated by the SESSL syntax. For example, OMNeT++ facilitates the specification of observation variables by pattern matching (note the wildcard asterisks in l. 7 and 12–16, Figure 9) and by supporting probability distributions (see l. 16). OMNeT++ variable assignments may also include time units (l. 14–16), so that the `range` function is used here to generate a sequence of strings.

[7]Java-based model entities are supported in OMNeT++ via `jsimplemodel`, see [Varga 2011, p. 9].

The first model parameter shall be scanned for two values (l. 13). The other three parameters (l. 14–16) shall be changed in unison, so they are joined together by an '`and`' function (l. 14–15), which is provided by the SESSL core. Thus, to specify a full-factorial experiment instead, one would simply replace the '`and`' invocations with commas. All parameters that are grouped via '`and`' must have the same number of values to scan. This is checked at runtime by the SESSL core. There can be arbitrarily many parameters, and they can be combined arbitrarily in this manner. The corresponding `omnetpp.ini` file generated by the specification in Figure 9 is shown in the Supplementary Material, Section B.

Future versions of this binding could support additional language elements to make the specification of observables, time units, and probability distributions less error-prone than using strings. Inspiration for such elements could be drawn, for example, from recent work on instrumentation languages [Helms et al. 2012]. The OMNeT++ binding also exposes some additional features like recording a complete event log (via an additional trait, l. 5) or removing observations that fall within the warm-up period (l. 10). Also note that both an executable and a network description file have to be given as a 'model' (l. 6). However, there are several other interesting features for specifying OMNeT++ experiments that are not yet supported, e.g., constraints on the parameter space (see [Varga 2011, p. 240 et sqq.]).

3.3.3. Opt4J & SSJ. The meta-heuristic optimization framework Opt4J [Lukasiewycz et al. 2011] offers several methods for simulation-based optimization, e.g., evolutionary algorithms and simulated annealing. To facilitate the integration of such tools, we developed a generic optimization interface for SESSL. The interface allows us to use optimization software with *any* simulation system that provides a SESSL binding, since they are all accessible through a common software layer. This saves the effort to develop optimization support for each simulation system individually. Our optimization interface retains maximal flexibility regarding the question of *what* to optimize, but is not yet feature-complete.[8]

Other simulation-related tools can be integrated via dedicated experiment facets, i.e., no additional SESSL interfaces are required. This allows, for example, to provide support for individual methods from software like the Stochastic Simulation in Java (SSJ) library [L'Ecuyer et al. 2002]. We integrated SSJ's methods for function approximation by providing an SSJ-specific experiment facet called `SSJOutputAnalysis`. Additionally, we integrated its methods for generating various kinds of LaTeX-based charts, by providing an implementation of the `AbstractReport` trait (see Figure 1), similar to `sessl.james.Report` (see Section 3.3.1).

Figure 10 shows an adapted experiment from [Ewald et al. 2010], but now specified with SESSL and using three software systems (not just JAMES II): it relies on JAMES II for simulation, on Opt4J for multi-objective optimization, and on SSJ for function approximation. The first few lines (l. 1–5) of the experiment import the SESSL core (l. 1), the generic optimization interface (l. 2), and the bindings for JAMES II, Opt4J, and SSJ (l. 3–5). Line 8 invokes the `optimize` function from SESSL's optimization interface, to declare the objectives `runtime` and `error`, both of which shall be minimized (`min`, l. 8). Additionally, an objective function (l. 9–31) is passed to the `optimize` function. It takes two arguments: a container with its parameter values (`params`, l. 8) and a container to store its value(s) (`objectives`, l. 8). The objective function executes a SESSL experiment (l. 10–30) based on the values in `params` (e.g., see l. 14) and stores the relevant results in `objectives` (e.g., see l. 24).

How to solve the optimization problem is specified by invoking the method `using` (l. 32), also provided by SESSL's optimization interface. It accepts a configuration for some optimization tool (here: Opt4J, l. 33) and applies it to the given problem. Opt4J is configured to use an evolutionary algorithm (l. 38), to print the results of the optimization (l. 39), and

[8]For example, one could add support for constraints, see [Law 2006].

```
 1 import sessl._              // SESSL core
 2 import sessl.optimization._ // SESSL support for simulation-based optimization
 3 import sessl.james._        // JAMES II binding
 4 import sessl.opt4j._        // Opt4J binding
 5 import sessl.ssj._          // SSJ binding
 6
 7 //Minimize both run time and error:
 8 optimize(("runtime",min),("error",min)) { (params, objectives) =>
 9   execute {
10     new Experiment with Observation with DataSink
11        with PerformanceObservation with SSJOutputAnalysis {
12
13       model = "file-sr:/./SimpleModel.sr" // Basic setup
14       set("r1" <~ params.get("synthRate"))
15       stopTime = 100000
16
17       observe("A") // Model instrumentation
18       observeAt(range(10000, 1000, 99000))
19
20       dataSink = MySQLDataSink(schema = "test_experiment") // Data storage
21       simulator = TauLeaping(epsilon = params.get("eps"))  // Simulation algorithm
22
23       withRunPerformance { perf => // Store runtime to objectives
24         objectives("runtime") <~ perf.runtime
25       }
26       withRunResult { result =>    // Calculate error with SSJ, store to objectives
27         objectives("error") <~ Misc.rmse(result.trajectory("A"),
28          fitAndEval(result.trajectory("A"), Polynomial(params.get("deg"))))
29       }
30     }
31   }
32 } using {
33   new Opt4JSetup {
34     param("synthRate", 1.0, 10.0) // Optimization parameters may refer to model,
35     param("eps", 0.01, 0.09)      // simulator, or
36     param("deg", 1, 4)            // output analysis (for example).
37
38     optimizer = EvolutionaryAlgorithm(generations = 20, alpha = 30)
39     withOptimizationResults { r => println(r) } // Print results to stdout
40   }
41 }
```

Fig. 10. Using Opt4J for the multi-objective optimization of a JAMES II simulation, and SSJ to calculate the objective.

to optimize three kinds of parameter at once (l. 34–36): `synthRate` is a model parameter (l. 14), `eps` is a simulator parameter (l. 21), and `deg` is a parameter for the result analysis (l. 28). The result analysis (l. 23–29) consists of a result handler to observe the simulator's run time, which is one optimization objective (l. 24), and a result handler for the observed simulation output, which calculates the second optimization objective (l. 26–29). The second objective, `error`, is defined as the root-mean-square error (RMSE) between the observed trajectory of species `A` and its `deg`-th degree polynomial fit (l. 27–28). While this example is a little contrived (no need to optimize `deg`, the fit of a polynomial can only improve as its degree increases), it shows the flexibility of specifying simulation-based optimization experiments with SESSL.

```
execute {
 new Experiment {
  // (SESSL constructs...)
  james2Experiment.setBackupEnabled(true) // Direct call to JAMES II API
 }
}
```

Fig. 11. Mixing SESSL constructs and custom code. The method `james2Experiment` enables direct access to the experimentation API. In contrast to general SESSL constructs, which are provided by the SESSL core, this method is defined in the JAMES II binding.

SSJ is invoked by the method `fitAndEval` (l. 28), which is provided by the trait `SSJOutputAnalysis` (l. 11) from its SESSL binding. Additional case classes like `Polynomial` are also provided by the binding, so that users can specify which approximation form to use. This kind of integration is easy to generalize: tools for specific experiment-related tasks (e.g., numerical analysis, statistics) could simply provide a new trait that can be mixed into SESSL experiments. If multiple tools provide similar interfaces for which no abstract trait exists yet, an abstract trait for this experiment facet should be included in SESSL's core, thereby growing the language (see Section 2.1).

Finally, the sample experiment also relies on the experiment facet `DataSink` (l. 10), which specifies that all observed simulation output should also be stored. For this, an experimenter must configure the data sink to be used (l. 20). A data sink is represented like any other SESSL algorithm (see Section 2.3.3). Here, the experimenter chooses a MySQL database. While a database schema is specified, the uniform resource locator (URL) and user credentials are not set explicitly, i.e., default parameter values are used (see Section 2.2).

3.3.4. Integration Cost. Although SESSL brings many useful features, a new binding has to be developed for each simulation tool. Developing such a binding for SBMLsimulator and OMNeT++ required about two and four single-developer workdays, respectively. Adding support for Opt4J took about three days, and adding support for function approximation and reporting via SSJ took about one day. We were not familiar with these software systems on a technical level. The time spans thus include understanding the APIs, developing the bindings as such, and specifying test experiments.

These efforts are comparatively small, considering their gains. For example, to the best of our knowledge SBMLsimulator does currently not support the configuration of parameter scans or optimization experiments, but both is now possible via SESSL. We could also easily realize a parallel execution of SBMLsimulator's runs and improved the observation of simulation output by automatically discarding irrelevant recorded data. Also, as mentioned before, SBMLsimulator output can now be included in result reports, with JAMES II or SSJ. The other software systems benefit from SESSL in a similar manner, particularly with respect to their interoperability.

Still, the current version of the SBMLsimulator binding merely consists of five files, containing 250 lines of code (LoC) overall. This brevity is made possible by several auxiliary data structures and default implementations provided by SESSL. For OMNeT++, on the other hand, no Java-compatible programming interface was available, which increased the complexity of the binding (ca. 500 LoC), even though it relies on the same auxiliary data structures as the SBMLsimulator binding. The bindings for Opt4J and SSJ consist of ca. 300 LoC and 100 LoC, respectively.

3.4. Customized Usage

As argued in Section 1, a major advantage of using an embedded DSL as a software layer between users and simulation systems is that experiments can be complemented by custom code. To realize this in SESSL, binding developers just need to make the actual experimentation API of the simulation system accessible. For JAMES II, this is done by providing a method `james2Experiment` that returns the JAMES II-specific object representing a simulation experiment. The code in Figure 11, for example, configures the ad-hoc experiment backup mechanism of JAMES II.

Accessing system-specific functionality this way should not be considered good practice in general, as it hampers re-use. On the other hand, there are many situations where simulation system independence is less important than offering the flexibility of the original API. By allowing 'power users' to add custom API calls, SESSL does not get in their way when it comes to in-detail configuration of system-specific aspects, but still helps them to reduce the boilerplate code for setting up simulation experiments.

4. DISCUSSION

We now describe how SESSL meets our requirements and discuss related work on domain-specific languages and simulation experimentation.

4.1. Requirements Revisited

In Section 1, we argued that three issues hinder the establishment of standardized experiment descriptions: 1) 'cutting-edge' experiments beyond the standard cannot be expressed, 2) dedicated tools are required (e.g., experiment editors), and 3) experiments may be underspecified and thus not reproducible.

SESSL solves the first two problems and simplifies the third. The first problem is solved by implementing an *embedded* DSL, and thus allowing to add custom code (Section 3.4). The second problem is solved by implementing SESSL in Scala, a statically typed programming language with tool support, e.g., for the Eclipse IDE [Dragos et al. 2013]. Powerful tooling facilitates the adoption of a DSL [Zdun and Strembeck 2009, p. 32]. The third problem is simplified by making the experiment specification dependent on a *particular* binding, which must be declared by an `import` statement. Still, reproducibility cannot be guaranteed: the behavior of a binding may change across versions. However, by explicitly associating experiment specification and execution semantics (via `import` statements), this problem is reduced to the well-known issue of dependency management. It can be solved, for example, with software repositories. SESSL experiments can also be re-used across systems, by specifying them in a system-independent manner (see Section 3.1). However, two bindings may interpret an experiment specification differently. System-independent test batteries can help to detect these problems (see Section 3.1).

As mentioned in Section 2.1, SESSL should be easy to learn and use. We tried to strike a good balance between simplicity and conciseness of the syntax (see Section 2.4). Issues like result handling and algorithm configuration are resolved so that system-specific complexity is encapsulated in the binding (see Section 2.3). Experiment composition (see Section 2.3.1) allows users to learn SESSL step by step, as they only need to deal with relevant language constructs for the task at hand. Regarding conciseness, a SESSL specification of the JAMES II experiment presented in [Ewald et al. 2010] would be less than half as long, and even the OMNeT++ experiment in Figure 9 is shorter than its corresponding `omnetpp.ini` file (see Supplementary Material, Section B).

While using SESSL requires some learning, experimenters can rely on IDEs with automatic code completion and incremental compilation for immediate feedback. We did not yet evaluate the effectiveness of SESSL with a comparative user study. Future work could approach this similarly to [Sobernig et al. 2011], where several implementations based on framework APIs are compared with a DSL-based implementation (and it turns out that DSLs can reduce development complexity).

To further improve user friendliness, the SESSL core thoroughly inspects each experiment before execution. For example, it is checked whether all observation times fall within the specified simulation time interval (if this is known beforehand). Such checks make it hard to execute inconsistent experiments. Handling those issues is particularly relevant for embedded DSLs: while the additional features provided by the host language give more freedom to the user, they also increase the risk of making mistakes [Sloane 2008, p. 7].

SESSL is a stand-alone software, independent from any simulation system. It is executed on the Java virtual machine, so that other software running on this platform can be integrated easily (e.g., see Section 3.3.1). The SESSL core provides well-documented interfaces for binding developers and much of the general functionality. Developers communicate the capabilities of a binding by choosing which experiment facets to implement (see Section 2.3.2). By adding custom experiment facets, they may also help to evolve the language. Being part of an embedded DSL, each facet may even provide custom sub-DSL(s) without much integration effort [Zdun and Strembeck 2009, p. 29].

On the other hand, developing a new binding requires knowledge of Scala and SESSL, and ultimately depends on the complexity of the simulation system to be integrated. Also, integrating simulation software from another platform is more difficult (see Section 3.3.2) and some SESSL features, e.g., the ability of adding custom code (see Section 3.4), may be hard to realize. However, Java is one of today's most popular software platforms and integration across different platforms is a general problem. In principle, one could also re-implement SESSL for other platforms, with a (more or less) similar syntax.

4.2. Related Work

4.2.1. Domain-Specific Languages. Zdun and Strembeck [Zdun and Strembeck 2009] name several main DSL design decisions: the DSL development process (also see [Strembeck and Zdun 2009]), the concrete syntax style, and the question of whether to use an embedded or an external DSL. In their terminology, we started out with extracting SESSL from an existing system, the experimentation layer of JAMES II [Himmelspach et al. 2008; Ewald et al. 2010], and proceeded with a language model-driven development process. In the nomenclature of Fowler, this means we now take a *language-seeded approach*: we sketch out how we would like to specify certain kinds of experiment and then define the concrete syntax as similar as possible [Fowler 2010, p. 41]. We chose a textual concrete syntax, as our target group includes developers, who often prefer it over a graphical syntax [Zdun and Strembeck 2009]. Moreover, textual experiment specifications are often easier to handle, e.g., version control systems are typically more convenient to use with simple text files. Finally, we chose to realize an *embedded* DSL.

Embedded DSLs have already been identified by others as a promising technique to facilitate scientific programming (e.g., [Hinsen 2013]), and are emerging for various domains, such as systems biology modeling (e.g., PySB [Lopez et al. 2013], embedded in Python), machine learning (e.g., OptiML [Sujeeth et al. 2011], embedded in Scala), and numerical analysis (e.g., Liszt [DeVito et al. 2011], embedded in Scala). Other DSLs embedded in Scala deal, for example, with database access [Garcia et al. 2010] or term rewriting [Sloane 2008]. Most Scala DSLs are complemented by compiler plug-ins, e.g., to generate platform-dependent code [DeVito et al. 2011]. Future versions of SESSL could do the same, e.g., to facilitate the integration of simulation systems from other platforms.

4.2.2. Simulation Experimentation. Historically, most DSLs for modeling and simulation have been powerful external DSLs, also known as *simulation (programming) languages* (e.g., see discussion in [Bruce 1997; Miller et al. 2010]), which provide features similar to general-purpose programming languages, but extend these by additional constructs. For example, in the realm of parallel discrete-event simulation, the APOSTLE language provides additional simulation-specific constructs, e.g., to declare time delays and control parallelization [Bruce 1997]. To the best of our knowledge, none of these languages is targeted at the domain of simulation *experiments*.

Simulation experimentation is often supported by either providing an API with a user-friendly syntax or an external DSL. For example, the simulation software STEPS is written in C/C++, but provides a Python interface to set up experiments [Hepburn et al. 2012], whereas OMNeT++ follows the second approach and offers an external DSL to set up simulation experiments (see Section 3.3.2).

Other approaches to facilitate simulation experimentation take a more holistic view. They strive to automate and document the whole process of conducting simulation studies in a certain domain, i.e., modeling, simulation, and data analysis, in order to avoid errors and improve reproducibility and credibility [Perrone et al. 2009]. This approach is realized by the SAFE framework, which is focused on automating experiments with the network simulator NS-3 [Perrone et al. 2012]. It uses the *ns-3 Experiment Description Language* (NEDL, see [Hallagan 2011]), which is based on the *Extensible Markup Language* (XML). NEDL supports factorial experiments, constraints on valid factor combinations, and termination conditions for single runs and replications (similar to `stopCondition` and `replicationCondition` in SESSL, see Supplementary Material, Section C). SAFE itself supports user management, a distributed execution of individual simulation runs, and simulation output storage and analysis. NEDL files are transferred to a remote execution service and then translated into C++ and Python code. SAFE provides a web-based interface for beginners, while power users can submit custom XML files and C++ simulation scripts via command line. In principle, SESSL should be easy to integrate into systems like SAFE, as it would simply provide an alternative to specify simulation experiments. When compared to the external DSL that is currently used in SAFE (NEDL), SESSL (currently) offers more features (e.g., support for performance experiments, simulation-based optimization) and the ability for power users to add custom language constructs on the fly ('growing' the language). Also, the tooling for both languages implies different transitions from beginner to power user: NEDL files are either generated via a web-based user interface or by manually editing XML code, while SESSL experiments gradually become more complex, by incorporating additional experiment facets. Finally, SESSL itself is system-agnostic and allows to integrate multiple tools, also for a single simulation experiment.

Systems like SakerGrid [Kite et al. 2011] or MEG [Page et al. 2012] are similar to SAFE in that their aim is to automate simulation experiment execution, but they are independent of a specific simulation system and focus on large-scale distributed experiment execution (e.g., on a grid). Both tools offer graphical user interfaces and some support for experiment design (e.g., MEG includes support for simulation-based optimization), but (to the best of our knowledge) they do not provide custom DSLs to set up simulation experiments. As with SAFE, SESSL could be used for this task.

To integrate different kinds of simulation software, frameworks like the Open Simulation Architecture (OSA) [Ribault 2011] allow a more principled and fine-granular reuse of functionality than SESSL. Here, the main focus is on reusing model and simulator components, but OSA can also be used to set up simulation experiments. This is done with an external DSL, i.e., XML configuration files for the software management tool Maven.[9] While this improves both the reproducibility and the reusability of experiments, the corresponding XML files are rather verbose [Ribault 2011, p. 104 et sqq.]. To support frameworks like OSA in SESSL, the complexity of these configuration files would be encapsulated by its binding.

A different approach towards simulation software integration is taken by tools like the Systems Biology Workbench (SBW) [Sauro et al. 2003], where various individual software systems communicate via binary messages. SBWs main task is to provide the functionality for service brokerage and (remote) method invocation. Therefore, its target audience are developers rather than experimenters, who will then use the results of the integration (e.g., a graphical model editor combined with a compatible simulator). SBWs architecture supports

[9]`http://maven.apache.org`

distributed computing and provides interfaces for multiple platforms, e.g., C/C++ and Java. Thus, a SESSL binding could use SBW's Java interface to control simulators already integrated into SBW. We consider this to be future work.

The systems biology community also endeavors to standardize simulation experiment descriptions. For example, the *simulation experiment description markup language (SED-ML)* [Waltemath et al. 2011] is an external DSL realized on top of XML. It is restricted to models encoded in XML, since the model elements to change (e.g., parameter values) are specified with XPath. An experiment description can involve multiple models and defines arbitrarily many simulation runs, so-called tasks, to be performed with them. Result reporting and observation are configured by declaring so-called data generators, which are associated with a specific task and may, for example, produce plots. The current SED-ML specification, Level 1 Version 2 [Bergmann et al. 2013], restricts simulation algorithms and their parameterization to entities defined in the KiSAO ontology [Courtot et al. 2011], which is focused on systems biology.

Future versions of SESSL could support SED-ML and other external DSLs for simulation experiments (e.g., NEDL), by supporting the import and export of experiment descriptions stored in this format. Moreover, these standardization efforts could benefit from the evolution of new language constructs in SESSL, as it provides a convenient test-bed for the iterative development of new description elements. At the same time, SESSL will also benefit from these efforts, as they represent a thoroughly discussed consensus of best practice in a (sub-)community. Following such agreements (e.g., regarding terminology) helps to prevent an unregulated growth of custom experiment facets with similar functionality.

5. CONCLUSIONS

We argue that standardized descriptions for simulation experiments are not yet established because 1) they do not allow for 'cutting-edge' experiments beyond the standard, 2) tool support is often lacking, and 3) experiments may still not be reproducible (see Section 1). We propose to address these problems by adding a separate software layer on top of simulation systems, in the form of an embedded domain-specific language. To illustrate the advantages of this approach, we introduce SESSL, a Scala-based domain-specific language for simulation experiments. After describing the rationale behind its design and syntax (Section 2), we illustrate its applicability (Section 3) and discuss its strengths in the context of related approaches (Section 4).

SESSL facilitates experiment reuse across simulation systems (Section 3.1), the runtime performance analysis of simulation (sub-)algorithms (Section 3.2), and other complex simulation experiments, such as simulation-based optimization (Section 3.3.3). It can also enable interoperability between components of different simulation systems (e.g., see Section 3.3.1). SESSL currently integrates five software systems: the simulation systems JAMES II, OMNeT++, and SBMLsimulator, as well as the optimization framework Opt4J and the simulation library SSJ. Bindings for other software systems are straightforward to implement (see Section 2.4.3).

We plan to broaden the scope of SESSL by supporting additional experiment facets and simulation systems in the future. To foster adoption in the M&S community, SESSL is open source (Apache 2.0 license). Its source repository is freely accessible at `http://sessl.org`.

6. ACKNOWLEDGMENTS

This research has been supported by the DFG (German Research Foundation), via research project `EW 127/1-1` (*ALeSiA*) and research training group 1387 (*dIEM oSiRiS*). We thank the anonymous reviewers for their helpful comments.

REFERENCES

BERGMANN, F. T., COOPER, J., LE NOVÈRE, N., NICKERSON, D., AND WALTEMATH, D. 2013. Simulation experiment description markup language (SED-ML): Level 1 version 2. http://sed-ml.org/documents/sed-ml-L1V2.pdf.

BRUCE, D. 1997. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In *Proceedings of the ACM SIGPLAN Workshop on Domain Specific Languages*. ACM, 17–35.

CAO, Y., GILLESPIE, D. T., AND PETZOLD, L. R. 2006. Efficient step size selection for the tau-leaping simulation method. *The Journal of Chemical Physics 124,* 4.

COURTOT, M., JUTY, N., KNUPFER, C., WALTEMATH, D., ZHUKOVA, A., DRAGER, A., DUMONTIER, M., FINNEY, A., GOLEBIEWSKI, M., HASTINGS, J., HOOPS, S., KEATING, S., KELL, D. B., KERRIEN, S., LAWSON, J., LISTER, A., LU, J., MACHNE, R., MENDES, P., POCOCK, M., RODRIGUEZ, N., VILLEGER, A., WILKINSON, D. J., WIMALARATNE, S., LAIBE, C., HUCKA, M., AND LE NOVERE, N. 2011. Controlled vocabularies and semantics in systems biology. *Molecular Systems Biology 7,* 1.

DESPEYROUX, T. 2008. Evolution of ontologies and types. In *Proceedings of the IADIS International Conference WWW/Internet 2008*, P. Isaías, M. B. Nunes, and D. Ifenthaler, Eds. 419–422.

DEVITO, Z., JOUBERT, N., PALACIOS, F., OAKLEY, S., MEDINA, M., BARRIENTOS, M., ELSEN, E., HAM, F., AIKEN, A., DURAISAMY, K., DARVE, E., ALONSO, J., AND HANRAHAN, P. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. ACM.

DRÄGER, A., KELLER, R., DÖRR, A., TSCHERNECK, S., HOFMANN, U., WRZODEK, C., FUNAHASHI, A., TABIRA, A., KANDEL, B., KLEIN, M., THOMAS, M., RODRIGUEZ, N., LE NOVÈRE, N., ZANGER, U. M., AND ZELL, A. 2012. SBMLsimulator: An efficient java solver implementation for SBML. http://www.ra.cs.uni-tuebingen.de/software/SBMLsimulator, accessed 12/2013.

DRAGOS, I., ODERSKY, M., BOURLIER, L., DOTTA, M., FARWELL, M., MILLER, H., MOLITOR, E., PLOCINICZAK, H., RUSSELL, M., AND STOCKER, M. 2013. ScalaIDE for Eclipse. http://scala-ide.org, accessed 12/2013.

EWALD, R. 2011. *Automatic Algorithm Selection for Complex Simulation Problems*. Vieweg + Teubner.

EWALD, R., HIMMELSPACH, J., JESCHKE, M., LEYE, S., AND UHRMACHER, A. M. 2010. Flexible experimentation in the modeling and simulation framework JAMES II–implications for computational systems biology. *Briefings in Bioinformatics 11,* 3, 290–300.

FOWLER, M. 2010. *Domain-Specific Languages* 1st Ed. Addison-Wesley Professional.

GARCIA, M., IZMAYLOVA, A., AND SCHUPP, S. 2010. Extending scala with database query capability. *Journal of Object Technology 9*, 45–68.

GIBSON, M. A. AND BRUCK, J. 2000. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *The Journal of Chemical Physics 104*, 1876–1889.

HALLAGAN, A. W. 2011. The design of XML-based model and experiment description languages for network simulation. Bachelor's thesis, Department of Computer Science, Bucknell University.

HELMS, T., HIMMELSPACH, J., MAUS, C., RÖWER, O., SCHÜTZEL, J., AND UHRMACHER, A. M. 2012. Toward a language for the flexible observation of simulations. In *Proceedings of the 2012 Winter Simulation Conference*, C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, Eds. IEEE.

HEPBURN, I., CHEN, W., WILS, S., AND DE SCHUTTER, E. 2012. STEPS: efficient simulation of stochastic reaction-diffusion models in realistic morphologies. *BMC Systems Biology 6*, 36.

HIMMELSPACH, J., EWALD, R., LEYE, S., AND UHRMACHER, A. M. 2010. Enhancing the scalability of simulations by embracing multiple levels of parallelization. In *Proceedings of the Second International Workshop on High Performance Computational Systems Biology (HiBi'10)*. IEEE, 57–66.

HIMMELSPACH, J., EWALD, R., AND UHRMACHER, A. M. 2008. A flexible and scalable experimentation layer. In *Proceedings of the 2008 Winter Simulation Conference*, S. Mason, R. Hill, L. Moench, and O. Rose, Eds.

HIMMELSPACH, J. AND UHRMACHER, A. M. 2007. Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium*. IEEE CS, 137–143.

HINSEN, K. 2013. A glimpse of the future of scientific programming. *Computing in Science & Engineering 15,* 1, 84–88.

HUCKA, M. ET AL. 2003. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics 19,* 4, 524–531.

HUDAK, P. 1996. Building domain-specific embedded languages. *ACM Computing Surveys 28,* 4es, 196.

JOPPA, L. N., MCINERNY, G., HARPER, R., SALIDO, L., TAKEDA, K., O'HARA, K., GAVAGHAN, D., AND EMMOTT, S. 2013. Troubling trends in scientific software use. *Science 340,* 6134, 814–815.

KITE, S., WOOD, C., TAYLOR, S. J. E., AND MUSTAFEE, N. 2011. SakerGrid: simulation experimentation using grid enabled simulation software. In *Proceedings of the 2011 Winter Simulation Conference*, S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, Eds.

LAW, A. 2006. *Simulation Modeling and Analysis* fourth Ed. McGraw-Hill Publishing Co.

L'ECUYER, P., MELIANI, L., AND VAUCHER, J. 2002. SSJ: A framework for stochastic simulation in Java. In *Proceedings of the 2002 Winter Simulation Conference*, E. Yücesan and C. H. Chen, Eds. IEEE.

LEISCH, F. 2002. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat 2002 - Proceedings in Computational Statistics*. Physica Verlag, Heidelberg, 575–580.

LI, C., DONIZELLI, M., RODRIGUEZ, N., DHARURI, H., ENDLER, L., CHELLIAH, V., LI, L., HE, E., HENRY, A., STEFAN, M. I., SNOEP, J. L., HUCKA, M., LE NOVÈRE, N., AND LAIBE, C. 2010. BioModels Database: An enhanced, curated and annotated resource for published quantitative kinetic models. *BMC Systems Biology 4*, 92.

LOPEZ, C. F., MUHLICH, J. L., BACHMAN, J. A., AND SORGER, P. K. 2013. Programming biological models in Python using PySB. *Molecular systems biology 9,* 1.

LUKASIEWYCZ, M., GLASS, M., REIMANN, F., AND TEICH, J. 2011. Opt4J: a modular framework for meta-heuristic optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. GECCO '11. ACM, 1723–1730.

MERALI, Z. 2010. Computational science: ...error. *Nature* 467, 775–777.

MILLER, J. A., BARAMIDZE, G. T., SHETH, A. P., AND FISHWICK, P. A. 2004. Investigating ontologies for simulation modeling. In *Proceedings of the 37th Annual Simulation Symposium*. IEEE CS.

MILLER, J. A., HAN, J., AND HYBINETTE, M. 2010. Using domain specific language for modeling and simulation: ScalaTion as a case study. In *Proceedings of the 2010 Winter Simulation Conference*, B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, Eds. IEEE.

ODERSKY, M., SPOON, L., AND VENNERS, B. 2011. *Programming in Scala* 2nd Ed. Artima.

ODERSKY, M. AND ZENGER, M. 2005. Scalable component abstractions. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, 41–57.

PAGE, E. H., LITWIN, L., MCMAHON, M. T., WICKHAM, B., SHADID, M., AND CHANG, E. 2012. Goal-Directed Grid-Enabled computing for legacy simulations. In *IEEE International Symposium on Cluster Computing and the Grid*. IEEE, 873–879.

PAWLIKOWSKI, K., JEONG, H. D. J., AND LEE, J. S. R. 2002. On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine 40,* 1, 132–139.

PERRONE, L. F., CICCONETTI, C., STEA, G., AND WARD, B. C. 2009. On the automation of computer network simulators. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. SIMTUTools '09. ICST.

PERRONE, L. F., MAIN, C. S., AND WARD, B. C. 2012. SAFE: simulation automation framework for experiments. In *Proceedings of the 2012 Winter Simulation Conference*, C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, Eds. IEEE.

RIBAULT, J. 2011. Reuse and scalability in modeling and simulation software engineering. Ph.D. thesis, Université de Nice Sophia-Antipolis.

SAURO, H. M., HUCKA, M., FINNEY, A., WELLOCK, C., BOLOURI, H., DOYLE, J., AND KITANO, H. 2003. Next generation simulation tools: The systems biology workbench and BioSPICE integration. *OMICS: A Journal of Integrative Biology 7,* 4, 355–372.

SLOANE, A. M. 2008. Experiences with domain-specific language embedding in scala. In *Proceedings of the 2nd International Workshop on Domain-Specific Program Development*, J. Lawall and L. Reveillere, Eds.

SOBERNIG, S., GAUBATZ, P., STREMBECK, M., AND ZDUN, U. 2011. Comparing complexity of API designs: an exploratory experiment on DSL-based framework integration. In *Proceedings of the 10th ACM international conference on Generative Programming and Component Engineering*. GPCE '11. ACM, 157–166.

STREMBECK, M. AND ZDUN, U. 2009. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience 39,* 15, 1253–1292.

SUJEETH, A., LEE, H., BROWN, K. J., ROMPF, T., CHAFI, H., WU, M., ATREYA, A., ODERSKY, M., AND OLUKOTUN, K. 2011. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 609–616.

VAN DEURSEN, A., KLINT, P., AND VISSER, J. 2000. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices 35,* 6, 26–36.

VARGA, A. 2001. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM'2001)*. SCS Europe.

VARGA, A. 2011. *OMNeT++ User Manual Version 4.2.2.* OpenSim Ltd. Last accessed 12/2013, http://www.omnetpp.org/doc/omnetpp/Manual.pdf.

WALTEMATH, D., ADAMS, R., BERGMANN, F., HUCKA, M., KOLPAKOV, F., MILLER, A., MORARU, I., NICKERSON, D., SAHLE, S., SNOEP, J., AND LE NOVERE, N. 2011. Reproducible computational biology experiments with SED-ML - the simulation experiment description markup language. *BMC Systems Biology 5*, 198.

WEBER, M. AND KINDLER, E. 2003. The petri net markup language. In *Petri Net Technology for Communication-Based Systems*, H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, Eds. Lecture Notes in Computer Science Series, vol. 2472. Springer, 124–144.

ZDUN, U. AND STREMBECK, M. 2009. Reusable architectural decisions for DSL design: Foundational decisions in DSL development. In *Proceedings of the 14th European Conference on Pattern Languages of Programs.*