Proceedings of the 2014 Winter Simulation Conference
A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, eds.

PERSPECTIVES ON LANGUAGES FOR SPECIFYING SIMULATION EXPERIMENTS

Johannes Schützel
Danhua Peng
Adelinde M. Uhrmacher

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

L. Felipe Perrone

Department of Computer Science
Bucknell University
1 Dent Drive
Lewisburg, PA 17837, USA

ABSTRACT

Domain specific languages have been used in modeling and simulation as tools for model description. In recent years, the efforts toward enabling simulation reproducibility have motivated the use of domain specific languages also as the means with which to express experiment specifications. In simulation areas ranging from computational biology to computer networks, the emerging trend is to treat the experimentation process as a first class object. Domain specific languages serve to specify individual sub-tasks in this process, such as configuration, observation, analysis, and evaluation of experimental results. Additionally, they can be used in a broader scope, for instance, to describe formally the experiment's goals. The research and development of domain specific languages for experiment specification explores all of these and additional possible applications. In this paper, we discuss various existing approaches for defining this type of domain specific languages and present a critical analysis of our findings.

1 INTRODUCTION

The fidelity of computational models for simulation and the accuracy of the simulators that execute them are determining factors for the quality of a simulation study. This motivates the rigorous practitioner to make strong efforts to create models at the appropriate level of abstraction and faithfulness, to validate them carefully, and to create corresponding verified computational implementations. Designing and executing the simulation experiment, however, are tasks in the workflow of the simulation process that are just as important and just as complex.

More often than not, the resulting product of the *design of experiment* (DOE) stage is a substantial amount of descriptive data that is used to drive the execution of the simulation study. These data include DOE entities such as *factors* (model parameters that are varied in the experiment design) and *levels* (specific values assigned to factors in the execution of the experiment), the DOE methodology used, and possibly resource identifiers for the computers involved in running the experiment, among other pieces of information. The complete description of the experiment, however, comprises much more information than just the DOE set up data. In general, it includes also a description of which data generated by the experiment are to be recorded and possibly additional records on control actions related to execution.

Keeping an accurate and complete record of all the conditions that define the experiment is of utmost importance to the reproducibility of the experiment. As observed by Pawlikowski, Jeong, and Lee (2002) and Kurkowski, Camp, and Colagrosso (2005) in the scope of network simulation, and by Merali (2010) and Joppa et al. (2013) in general scientific computing, having complete records of the experiment increases the scientific rigor of the experiments and, consequently, also their credibility. In the *Minimum Information About a Simulation Experiment* (MIASE) standard proposed by Köhn and Novère (2008), the authors

establish that in order for an experiment to be reproducible by third parties, the record of its execution must contain: (1) the composition of the model that is simulated together with its configuration parameters, (2) the simulation methods used (for instance, termination conditions), (3) the collection of tasks performed in the experiment, and (4) the complete collection of outputs that is produced.

Rahmandad and Sterman (2012) identify distinct reporting requirements for *model* and *simulation experiment*, which are divided into two levels: *minimum* and *preferred*. Their *Minimum Model Reporting Requirements* (MMRR) include descriptions of the computational operations in the model, a list of model default values and factors for experimentation, and information on the units of measure employed for all the model's default values and factors. The *Preferred Model Reporting Requirements* (PMRR) goes beyond the MMRR to include also information on the sources of data for the model's equations and algorithmic rules, the definition of all variables used in the model, and source code for the model's computational implementation. The *Minimum Simulation Reporting Requirements* (MSRR) speaks to the experiment and includes reporting of the simulation hardware and software platforms, the simulation algorithms used, pre-processing used to generate input data for the experiment, all the levels applied to factors in the simulation model, the number of iterations of the experiment, and all the post-processing performed on the output data. The *Preferred Simulation Reporting Requirements* (PSRR) adds information on random number generation and confidence levels used for estimation, among other requirements.

The evolution of these standards for enabling the reproducibility of simulation experiments sets a high bar for reporting requirements. A considerably large volume of data interlinked by non-trivial relationships is needed to provide a complete and accurate record of the simulation experiment. The challenges in dealing with these complex and abundant data can be effectively addressed with the design and the use of special purpose simulation *experiment specification languages* (ESL).

Although the concept of ESL is relatively new, *modeling languages* are well established in the field of modeling and simulation dating back to the 60s, when GPSS, SIMSCRIPT, and SIMULA were notable examples. Recent literature shows that this has been a very active area of research and development. However, it is worth remarking that to a great extent, few of the efforts in language design have aimed to reach comprehensively the goals enumerated in the MIASE, PMRR, and PSRR standards. In this paper, we explore some interesting general approaches in ESL development and focus our attention on two languages in which we have been directly involved.

2 TYPES OF MODELING AND SIMULATION SUPPORT LANGUAGES

2.1 Workflow Languages

Rigorous simulation studies follow a general sequence of actions that is well established and broadly known (Law 2007, Sargent 2013). There has been, however, a recent paradigm shift in understanding the modeling and simulation processes as just another incarnation of *scientific workflows*, which is opportune. With the fast evolution of computer support tools for empirical research in natural sciences (Altintas et al. 2003), the modeling and simulation community is now able to capitalize on the lessons learned in a related field. The use of workflow management systems in the modeling and simulation life-cycle enables a no-compromise, sensible balance between supporting the user in creating rigorous experiments and creating simulation systems that can be easily adapted to different scenarios. The workflow system can support the generation, the documentation, and the reproduction of simulation experiments.

With the use of workflows, the process of experimentation can be built into the tools by the system designer, as in (Reiter et al. 2012), or exposed to the simulationist as a series of templates to be completed, as in (Rybacki et al. 2012). Other benefits, as shown by Rybacki et al. (2014), may include the interweaving of modeling and experiment execution supported by an artifact-based approach and the ability to apply constraints to guide the experimental process.

There exist several established languages to define the structure of workflows. Some of these have a particular focus on business processes, such as BPEL/BPMN (Weidlich et al. 2008). Others were

created to support general scientific processes, such as Karajan (von Laszewski et al. 2006). In comparison, established domain-specific languages (DLSs) for experimenting with formal models (and possibly generating the corresponding workflows) appear still missing.

2.2 Domain-Specific Languages

A *domain-specific language* (DSL) is a programming language that is tailored specifically for an application domain. According to van Deursen, Klint, and Visser (2000), the DSL *"offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."* There are two basic types of DSLs: *internally defined* (or *embedded*) and *externally defined*. The internally defined languages are implemented using constructs of a general-purpose programming language, which is called its *host language*. For this reason, programmers who are already proficient in the host language will likely find this type of DSL easy to learn. Another benefit is that they require minimal implementation effort on the language designer. Most importantly, internally defined domain-specific languages can easily be extended by users. In contrast, externally defined languages give their designer complete freedom to construct it, even though they require new interpreters or compilers. Their major advantage is that since they can be designed from the ground up, they may be more direct, compact and expressive, since they are not encumbered by any host language. It is acknowledged that the directness and compactness of externally defined DSLs can also make it easier to learn. Although most DSLs used in modeling and simulation can be classified as externally defined, embedded DSLs for this purpose also exist (Miller et al. 2010).

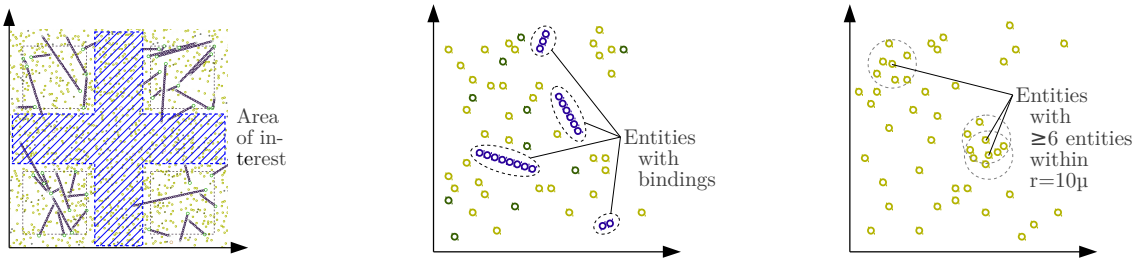
2.3 Languages to Instrument Data Collection

If all the model state trajectories of all simulation runs in an experiment were to be recorded, the volume of data accumulated could easily become overwhelming. For this reason, it is important to consider strategies to reduce the volume of data. One way to achieve this goal is through targeted data extraction, which is achieved by instrumenting the model according to the experiment's goals defined by the user. Many modeling and simulation software products offer mechanisms for specifying which data is to be observed. The approaches we have identified include visual mechanisms, that is, selecting variables in a GUI, extensions to the model description, and instrumenting observation via coding. In OSIF (Ribault et al. 2010), for example, support for aspect-oriented programming in the host language allows the weaving of observation code into the models' implementations. In contrast, Helms et al. (2012) present an external domain-specific instrumentation language for the modeling language ML-Rules. What these two approaches have in common is that they emphasize the importance of treating observation and instrumentation separately from modeling, as a first class abstraction in the experimental process.

Within this context of separation between modeling and data collection code, the design goals of an instrumentation language for data collection include the following. First, for the sake of effective data analysis, there must be support for *addressing* specific parts, that is *targets*, within the model. Second, there must be support for the application of functions to individual entities and/or groups of entities within the model so that higher-level information such as metadata can be obtained and aggregated. Finally, there must be support for the collection of data during specific instants and periods of simulation time, as well as in specific phases of the state-space of the experiment.

Target Addressing: In order to collect data about a given model's state, one needs to be able to identify or to address the entities within the model that contain the data of interest. In dealing with structured models that comprise entities characterized by attributes, addressing strategies may consider conditions on the entity's position within the structure, conditions on the values of attributes, conditions on a broader context, or combinations of the aforementioned. The actual structure of the model determines what precisely needs to be expressed in terms of structure-based and context-based addressing. In the following, we discuss spatial structures and hierarchical structures.

When dealing with entities in 2D or 3D space, as in an ML-Space model (Bittig et al. 2011), one might address: areas or volumes of interest (Fig. 1a), entities identified by constraints on attributes (Fig. 1b), or entities identified by conditions that constrain the context (Fig. 1c). How to address targets depends largely on the spatial modeling language, whether space is discretized or continuous, and whether or not excluded volumes and/or nesting are considered (Bittig and Uhrmacher 2010).



(a) Position-based. (b) Attribute-based. (c) Context-based.

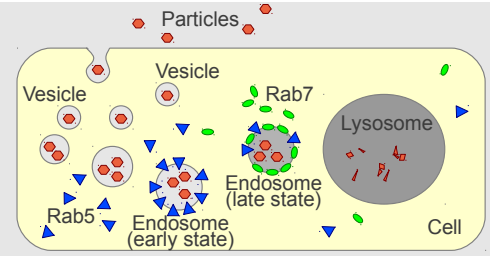
Figure 1: Addressing of entities in spatial models.

In models with a hierarchical structure, such as those in ML-Rules (Maus et al. 2011), DEVS (Zeigler 1984), and ns-3 (Riley and Henderson 2013), one might want to address entities by *paths* within a name space. Addressing schemes may support the use of constraints on attributes and possibly also *wildcards* on path elements, an approach similar to that of the XPath language for querying XML documents (W3C 1999). XPath allows one to specify paths, where at each hierarchy level, the set of selected nodes can be constrained by the nodes' non-unique names and/or by predicates on the nodes' attributes, or on contexts defined by other XPath expressions. The definition of predicates may use boolean, arithmetic, and comparison operators, as well as functions on strings and sets.

In hierarchical organizations such as those in DEVS models, patterns may be defined in terms of paths along communication relations rather than on the hierarchy. For a DSL aimed at instrumenting structured network models, query languages for graph-based databases may serve as a starting point (Wood 2012). These languages, which support the expression of relations among paths, are of particular interest if the network structure changes dynamically, e.g. address targets, A and B, such that the same sequence of edge labels connects A and B as it connects C and B.

Data Collection: In addition to selecting the entities of interest in a model, the language must allow one to select the values from the entities that are to be externalized in simulation output. In certain cases, it may be sufficient to denote an entity's attribute from which the value is to be collected. When one or more attribute values need to be subjected to functions for transformation or aggregation, the language should allow the expression of operations like $f\big(val(attr)\big)$ or $f\big(val(attr_1),\ldots,val(attr_n)\big)$, respectively. When a set of entities is addressed, multiple values may be selected, however, a plain list of values is often not the desired output as provenance information may need to be recorded and/or values may need to be grouped. In standard database queries, the extraction of information for identification or grouping and the extraction of the "target value" are part of the projection function. A data collection language ought to allow one to specify not only how groups of values are formed, but also the aggregation functions that should be applied to the values within each group.

Phases and Frequency of Data Collection: In many circumstances, the simulationist is interested in collecting data only from specific phases of the simulation, such as during transient or steady-state. For this reason, languages should allow one to specify the start and the end times of phases in which data should be collected. Another helpful language feature would be to allow the enabling and the disabling of data collection based on conditions on model attributes, such as *when((A.population<100) && (B.rate>1.5).*



(a) Modeling the endocytosis process of cells requires a modeling approach that supports dynamic hierarchical structures such as ML-Rules as endosomes form at the membrane and might fuse with lysosomes in the cytosol.

```
new Instrument {
  start at SimTime(100.0)
  stop at SimEnd     // optional because SimEnd is default
  trigger atInterval 10.0    // short for SimTime(10.0)
  filter accept (name is "Particle") &
       compartment(name is "Endosome")
  filter reject compartment(attrib(1) is
       "late")
  extract (quantity as "q")
  aggregate (sum("q"))  // implicit grouping (all in one)
}
```

(b) Example of instrumentation. The language takes into account that entities are named (`name`), located in a hierarchy (`compartment`), and have attributes (`attrib`), which are accessed by indices (`1`).

Figure 2: Specifying how a model with dynamic hierarchical structures can be instrumented (using a Scala-based embedded DSL).

(Clearly, this type of feature requires the appropriate target addressing capability.) Finally, the language may include mechanisms to request data collection in discrete instants of simulation time or according to some pre-determined frequency.

2.4 Languages to Support Validation and Analysis

According to Cellier's definition *"A model M for a system S and an experiment E is anything to which E can be applied to answer questions about S"* (Cellier 1991). However, not all experiments are conducted to answer questions about a system. Simulation experiments may be conducted to answer questions about the model itself and often in relation to the system. The questions refer typically to behavioral properties which are inferred from sequences of output values (that is, *trajectories*) produced in the experiment. Languages that support validation and data analysis provide formalisms that allow for the automation of model checking by verifying whether required properties of the model are satisfied during the simulation run.

What to express: The evaluation of whether models exhibit the desired behavioral properties involves the analysis of the trajectories produced in the experiment. The questions asked of the model determine whether multiple configurations of simulation experiments may be needed. Stochastic models, for example, require multiple replications in order to enable sound output data analysis. In general, model validation and data analysis may require a single replication, multiple replications for one configuration, or multiple configurations. Within one replication, one variable or multiple variables may need to be checked as in the example: *prey should either not become extinct or its population should be larger than that of predators*. These kinds of properties form the basis of more complex temporal properties, which relate to basic properties over time, as in the example: *relatively short stimulations (5 minutes or less) by tumor necrosis factor α can trigger sustained activation (60 minutes) of the NF-B pathway in IB-NF-B signaling model* (Prill et al. 2005). Temporal properties may refer to the dynamics of one or more variables, as in the example of the prey-predator model: *predators start increasing only after prey density grows beyond a threshold*. Another common property of interest is whether a simulation exhibits cyclic behavior, which motivated the study of specific algorithms for detecting cycles in time series. Lastly, a property could be stated to hold for a length of time or a specific bounded time interval, as in the example: *after the removal of the growth factors, the concentration of beta catenin should rise for 1 hour*.

For stochastic models, it is important to represent the probability that the experiment's results meet a certain property. One may define the probabilities with which properties should hold at discrete points or intervals of time over one same trajectory and across different replications of a single configuration. For

example, *"some properties hold at least 80% of the time in a certain interval"* or *"some properties hold in more than 8 replications out of 10"*. There could also be nested probabilities to express complex properties, such as *"in more than 8 replications out of 10, some properties hold at least 80% of the time in a certain interval"*. It would also be interesting to express properties that span different experiment configurations, as in the example: *the so called ratio-dependent theory states that if an ecosystem has richer resources, there should be higher equilibrium abundances on all trophic levels, in comparison to an ecosystem with less resources* (Ginzburg and Akçakaya 1992). Finally, experiments focused on sensitivity analysis and robustness depend on comparing the results of different configurations, so single configuration experiments are not sufficient to provide adequate information on a model's behavior.

Approaches to express properties: In the verification and model checking, there are typically three approaches to express properties which need to be satisfied by a model: logic-based approach, automata-based approach and the hybrid approach of logic and automata (Bouajjani and Lakhnech 1996).

The logic-based approach uses languages to specify properties. For time related properties, various temporal logics exist, such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL). LTL provides the "next" and "until" operators from which the temporal connectives "sometimes" and "always" in the future can be derived, in addition to the standard logic connectives. CTL allows one to express non-determinism in future behavior and provides two additional path qualifers: "existential" and "universal". Although LTL and CTL support deterministic models, Probabilistic Computation Tree Logic (PCTL) and Continuous Stochastic Logic (CSL) extend CTL to support discrete- and continuous-time stochastic models, respectively. In PCTL and CSL, the probabilistic path quantifier P is introduced in place of the universal path quantifier A (for all paths) and the existential path quantifier E (there exists a path such that) (Tomita et al. 2011).

Automata are often used as an alternative to temporal logic for specifying properties, e.g., Büchi automata (Alpern and Schneider 1989). Regular expressions and their extensions are also useful formalisms (Barringer et al. 2004). It has been shown that automata and regular expressions have at least the same power of expression as temporal logics. The advantages and disadvantages of these approaches are difficult to judge without knowledge of the type of properties to be described. However, in general, automata and regular expressions are perceived as too procedural and low-level to be attractive specification formalisms (De Giacomo and Vardi 2013).

Consequently, logic-based approaches tend to prevail in specifying properties for (statistical) model checking. For example, in (Rizk et al. 2009) to ensure the robustness of a synthetic genetic network, the expected property, i.e., that the fluorescence remains below 10^3 for at least 150 min, exceeds 10^5 after at most 450 min, and needs for switching from low to high levels at most 150 min, by a temporal formula: $\phi(t_1, t-2) = G(time < t_1 \rightarrow X < 10^3) \wedge G(time > t_2 \rightarrow X > 10^5) \wedge t_1 < 150 \wedge t_2 < 450 \wedge t_2 - t_1 < 150$. The approach is based on deterministic execution of models. For stochastic models, in (Peng et al. 2014), the property ϕ that the prey die out first and then the predators die out, has been expressed in LTL $((\#Prey = 0)\ R\ (\#Predator > 0)) \wedge F\ (\#Predator = 0)$ and to take the probability into account, the property tested was denoted as $Pr_{\geq 0.8}(\phi)$. Using the *Simulation Experiment Specification via a Scala Layer* (SESSL), which is discussed in the next section, this property was expressed as:

```
assume( Pr(((Negation(variable("Prey") > 0)) R (variable("Predator") > 0))
            and (Negation(G(variable("Predator") > 0)))) >= 0.8)
```

Finally, up to the level of replications, properties can be and have been frequently expressed by standard temporal logic. However, comparing the result of different configurations, which is central for many experiments, requires the definition of additional operators that work on different paths with different starting points.

3 EXPERIMENT SPECIFICATION LANGUAGES

A special purpose language for specifying simulation experiments may share several aspects in common with the other types of languages that support modeling and simulation. First, it may borrow concepts and elements from the workflow languages discussed in section 2.1. An experiment may consist of a collection of *components* that encapsulate the tasks that need to be performed. The ESL should allow hierarchical compositions so that new components may be constructed as aggregations of existing ones. Additionally, the ESL should include mechanisms that allow the simulationist to express how components share data with one another. If every component may consume and produce data, the structure of the experiment is defined by a dataflow graph, which should be described by the ESL's syntactic elements.

Second, it stands to reason that the design of ESLs should be driven by the needs and the peculiarities of specific application domains, rather than aim to support all types of models and simulations. It is less important whether they are designed as internally or externally defined DSLs and more important that they support well the targeted application domain. Since the ESL should be easy to learn and to use, as discussed in section 2.2, the closer the language's affinity to the application domain, the more intuitive it will feel for the simulationist.

Third, because the goal of experiments is to generate data, it is important that the ESL provides mechanisms to define answers to various questions related to data collection: which data to collect, how much of them to collect, how often to collect, and what pre-processing to apply to data. As discussed in section 2.3, in order to support data collection activities, the language must include mechanisms to identify the elements within the model that contain the data of interest, to aggregate data from potentially multiple sources, to determine at which periods in simulation time to actively record data, and to specify the frequency of data collection.

Finally, ESL design should aim to support experimentation in two different stages of the M&S process: during model validation and after production runs with the model. Section 2.4 showed that in order to support validation, the ESL should be able to express properties of the model with respect to time, value constraints, and randomness.

The remainder of this section focuses on a few examples of ESLs. The first one, *Simulation Experiment Specification via a Scala Layer* (SESSL) is a general purpose language defined as an internally defined DSL. The next two, *ns-3 Experiment Description Language* (NEDL) and *SAFE Language for Experiment Description* (SLED) are specific to network simulation.

3.1 SESSL

We illustrate the features of SESSL (Ewald and Uhrmacher 2014) with an experiment from (Ewald et al. 2010). In this example, shown in Fig. 3, we use optimization to estimate the kinetic reaction constants (r_1 and r_2) of two reactions in a rather simple reaction species model. SESSL is system agnostic and facilitates experiment reuse across different simulation systems. Therefore, SESSL consists of two layers: `sessl._` refers to basic definitions and `sessl.james._` contains the bindings for the software system.

The specification example contains a reference to the model, defines the number of replications, the stop condition for executing the model, the goal, range, and method of optimization, data collection and storage, random number generator, and execution algorithm. A comparison with the solution shown in (Ewald et al. 2010) which has been realized in the experimental frame layer of JAMES II shows the benefit of a domain language in terms of succinctness of the needed code to generate an experiment.

The example also shows how SESSL can easily be extended by additional experiment facets. Therefore, Scala offers traits which are interfaces that may also contain method definitions and member variables, and therefore can implement own functionality. In the example the Experiment is enriched with features for observing, for parallel execution (in the example we are using the default setting, by adding `parallelThreads = -1` to the specification, we would have used all threads but one), for data sinks and for optimization. Here we use the optimization methods that are provided in JAMES II. Methods of other

```
import sessl._
import sessl.james._

new Experiment with Observation with ParallelExecution with DataSink with Optimization {
  model = "file-sr:/./SimpleModel.sr"
  replications = 10; stopTime = 100000
  optimizeFor("x" ~ "A")(result => result.max("x"))*
  optStopCondition =
    OptMaxAssignments(100) or OptMaxTime(hours = 1)
  optimize {
    "synthRate" ~ "r1", range(1.0, 10.0),
    "degradRate" ~ "r2", range(5.0, 15.0)}
  startOptimizationWith {
    "synthRate" <~ 1.0,
    "degradRate" <~ 5.0}
  optimizer = HillClimbing()*
  observeAt(10000, 20000, 99900)*
  dataSink = MySQLDataSink(schema = "test2")*
  rng = MersenneTwister()
  simulator = DirectMethod()
}
```

Figure 3: A SESSL experiment using JAMES II. (Scala keywords are shown in blue.)

software can be integrated. In (Ewald and Uhrmacher 2014) it is shown how different software systems can be linked into SESSL, i.e., how via `import sessl.ssj._` and `new Experiment ...with SSJOutputAnalysis` specific methods provided by the SSJ library for stochastic simulation can be used and how SESSL can serve as a unified interface for other tasks, such as simulation-based optimization (integrating the Opt4J framework for meta-heuristic optimization). As embedded language SESSL contains Scala constructs, e.g., `(result => result.max("x"))` defines an anonymous function. Clearly, the definition in SESSL is more compact and closer to our understanding of an experiment and thus can more easily be understood. However, to specify an experiment in SESSL requires also some knowledge about Scala, its syntax and semantics.

SESSL contains rudimentary constructs to collect data, i.e., defining the variables to be observed (`"A"`) and the frequency of observing them, (`observeAt(10000, 20000, 99900)`), other constructs refer to evaluating the results, e.g., `result.max("x")`. Thus, possibilities to include more sophisticated means for observation and analysis should be included. However, this can easily be realized by integrating specialized languages as additional traits, as has already been done in a first approach to support statistical model checking experiments in SESSL (Peng, Ewald, and Uhrmacher 2014).

It should be noted, that SESSL is a language for specifying and generating rather than for describing experiments. This is in contrast to community approaches such as the SED-ML Simulation Experiment Description Markup Language, which has been developed for exchanging, encoding, and documenting (Waltemath et al. 2011) experiments. Both languages however have in common that they are declarative and that they are not constrained to one specific simulation system. In contrast, the experimentation language NEDL has been developed for *ns-3*.

3.2 NEDL and SLED

In the development of the *Simulation Automation Framework for Experiments* (SAFE) (Perrone, Main, and Ward 2012), it became immediately clear that standardizing experiment descriptions was necessary in order to document scenarios and enable reproducibility. The *ns-3 Experiment Description Language* (NEDL) (Hallagan 2010) was created to meet the demand for a language capable of capturing all the information that defined the scenario of an experiment. The design goals of this language were threefold: it had to be *expressive*, *compact*, and *intuitive* to the simulationist. The information contained in a NEDL file specifies a design of experiment space in terms of factors, levels, and constraints that aim to prune away design

points that are beyond the interest of the simulationist. Since NEDL is based on XML, the language defines a collection of "elements," which may be either compulsory or optional in the description of an experiment. With NEDL syntax, the simulationist enumerates the factors of interest in the experiment and associates lists of levels to each one. The language provides various mechanisms to simplify the definition of lists of levels and also to constrain the range of valid levels for each factor.

Because NEDL is an externally defined DSL, it must be processed by special-purpose tools for parsing and document validation. A parsed NEDL document results in an unmarshalled XML object that is accessed by another tool (embedded in SAFE) that generates the set of points in the design of experiment space. This allows SAFE to process each design point sequentially and dispatch simulation runs to a collection of machines that collaborate on implementing the *multiple replications in paralell* execution paradigm.

The development of NEDL and its supporting tools showed promise in meeting the design goals of the language. Its practical application, however, turned out to be somewhat problematic. Although the language met the expressiveness objective, it turned out to be overly verbose (as XML-based languages tend to be). We addressed this shortcoming by developing form-like interfaces that collect information from the user and generate NEDL syntax automatically. This reduces the pressure on the simulationist and, at the same time, guarantees the well-formedness and the validity of the experiment description file.

NEDL documents, however, turned out to be complex and tricky to process, which complicated the development of a robust parser. Its `sequence` element for encoding arbitrary mathematical expressions to generate lists of factor levels serves to illustrate the problem. An expression as simple as $T = 10 \times i$, for $10 \leq T \leq 100$ can be written `T = [10*i for i in range(1,11)]` as a Python list comprehension. In NEDL syntax, this list turns into the code snippet shown in Fig. 4.

The lessons learned with NEDL motivated the creation of a replacement language called *SAFE Language for Experiment Description* (SLED). This new language, which is based on the JSON format, is simpler, smaller, and substantially more compact. Fig. 5 shows the syntactical elements in SLED, which are used to create files that contain the design of experiment space and also information on the experiment's code base installation, the name of the file containing the source code of the corresponding ns-3 script. This syntax allows one to associate with each factor some descriptive text, a default level, a range of valid levels, and also the `visible` annotation which configuration interfaces can inspect to determine which factors should be exposed to the user.

Even though SLED is also an externally defined DSL, the use of the JSON format makes it much easier to parse. Ongoing work on the development of SLED is exploring options to use Python list comprehensions directly, expressing relationships between experimental factors, and creating syntax for design of experiment space pruning that is more powerful than what is provided in NEDL. Additional developments in SLED will explore a connection to the ns-3 data collection framework presented by Perrone et al. (2013). This will enable the type of functionality discussed in section 2.3. In particular, it will record in the experiment specification file the periods in simulation time in which data collection takes place and the frequency of sample collection.

```
<sequence>
  <factor>T</factor>
  <test>EQUALS</test>
  <lvar>i</lvar>
  <op>MULT</op>
  <rconst>10</rconst>
  <where>
      <range>
          <var>i</var>
          <lo>1</lo> <hi>10</hi> <delta>1</delta>
      </range> </where>
</sequence>
```

Figure 4: A list comprehension for generating factor levels in NEDL.

```
{ "name": "ExpDescription Name",
  "description": "This is an example description",
  "script": "script name",
  "public": false,
  "factors": [{ "name": "First Factor",
                "help": "factor description goes here",
                "type": "float",
                "visible": true,
                "default": 1.4,
                "min": 0,
                "max": 100},
              { "name": "Second Factor",
                "help": "more factor description",
                "type": "boolean",
                "visible": false,
                "default": false}]
}
```

Figure 5: Elements of a SLED file.

4 CONCLUSION

In this paper, we explore some, though not all, of the current uses and possibilities for domain specific languages in simulation. Languages to support experimentation are becoming more important in the area because they allow compactness and expressivity in experiment description, thereby facilitating the experiments' execution and enabling their dissemination and reproduction. Each different task of the simulation workflow, such as experiment specification and data collection, may call for its own different specialized language. Subcomponents of these tasks may call for yet other specialized languages. Looking forward, we realize that it will be important that simulation frameworks provide the means of a hierarchical integration of these support languages into a cohesive whole. We also expect that it will be crucial to allow for flexibility and language extensibility in order to address the changing needs of applications.

ACKNOWLEDGMENTS

Johannes Schützel is supported by the German Research Foundation (Research Training Group 1424 "MuSAMA"). **Danhua Peng** is supported by the China Scholarship Council (CSC) and the National Natural Science Foundation of China (Grant No. 61374185). **L. Felipe Perrone** is supported by the U.S. National Science Foundation (NSF) under Grant Nos. CNS-0958139, CNS-0958142, and CNS-0958015. Any opinions, findings, and conclusions or recommendations expressed in this material are the authors' alone and do not necessarily reflect the views of the NSF. Felipe thanks William S. Stratton and Andrew W. Hallagan (Computer Science & Engineering students at Bucknell University) for contributing the material on SLED and NEDL used in this paper.

REFERENCES

Alpern, B., and F. B. Schneider. 1989, January. "Verifying Temporal Properties Without Temporal Logic". *ACM Transactions on Programming Languages and Systems* 11 (1): 147–167.

Altintas, I., S. Bhagwanani, D. Buttler, S. Chandra, Z. Cheng, M. Coleman, T. Critchlow, A. Gupta, W. Han, L. Liu, B. Ludascher, C. Pu, R. Moore, A. Shoshani, and M. Vouk. 2003. "A modeling and execution environment for distributed scientific workflows". In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, 247–250.

Barringer, H., A. Goldberg, K. Havelund, and K. Sen. 2004. "Rule-Based Runtime Verification". In *Verification, Model Checking, and Abstract Interpretation*, edited by B. Steffen and G. Levi, Volume 2937 of *Lecture Notes in Computer Science*, 44–57. Springer Berlin Heidelberg.

Bittig, A. T., F. Haack, C. Maus, and A. M. Uhrmacher. 2011. "Adapting Rule-based Model Descriptions for Simulating in Continuous and Hybrid Space". In *Proc. of the 9th International Conference on Computational Methods in Systems Biology*, CMSB '11, 161–170. New York, NY, USA: ACM.

Bittig, A. T., and A. M. Uhrmacher. 2010. "Spatial modeling in cell biology at multiple levels". In *Proc. of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 608–619. Piscataway, New Jersey: IEEE.

Bouajjani, A., and Y. Lakhnech. 1996. "Logics vs. automata: The hybrid case". In *Hybrid Systems III*, edited by R. Alur, T. Henzinger, and E. Sontag, Volume 1066 of *Lecture Notes in Computer Science*, 531–542. Springer Berlin Heidelberg.

Cellier, F. 1991. *Continuous System Modeling*. Springer.

De Giacomo, G., and M. Y. Vardi. 2013. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In *Proc. of the 23rd Int. Joint Conference on Artificial Intelligence*, 854–860: AAAI Press.

Ewald, R., J. Himmelspach, M. Jeschke, S. Leye, and A. M. Uhrmacher. 2010, May. "Flexible experimentation in the modeling and simulation framework JAMES II–implications for computational systems biology". *Briefings in Bioinformatics* 11 (3): 290–300.

Ewald, R., and A. M. Uhrmacher. 2014. "SESSL: A Domain-Specific Language for Simulation Experiments". *ACM Transactions on Modeling and Computer Simulation* 24 (2).

Ginzburg, L., and H. Akçakaya. 1992. "Consequences of ratio-dependent predation for steady-state properties of ecosystems". *Ecology* 73 (5): 1536–1543.

Hallagan, Andrew W. 2010. "The Design of XML-Based Model and Experiment Description Languages for Network Simulation". Honors Thesis, Bucknell University.

Helms, T., J. Himmelspach, C. Maus, O. Röwer, J. Schützel, and A. M. Uhrmacher. 2012. "Toward a language for the flexible observation of simulations". In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. Uhrmacher. Piscataway, New Jersey: IEEE.

Joppa, L. N., G. McInerny, R. Harper, L. Salido, K. Takeda, K. O'Hara, D. Gavaghan, and S. Emmott. 2013, May. "Troubling Trends in Scientific Software Use". *Science* 340 (6134): 814–815.

Köhn, D., and N. Novère. 2008. "SED-ML - an XML formate for the implementation of the MIASE guidelienes". In *Proc. of the 6th International Conference on Computational Methods in Systems Biology*, edited by M. Heiner and A. Uhrmacher, 176–190: Springer.

Kurkowski, S., T. Camp, and M. Colagrosso. 2005. "MANET Simulation Studies: The Incredibles". *ACM SIGMOBILE Mobile Computing and Communications Review* 9 (4): 50–61.

Law, A. 2007. *Simulation Modeling and Analysis*. 4th ed. New York: McGraw-Hill.

Maus, C., S. Rybacki, and A. M. Uhrmacher. 2011. "Rule-based multi-level modeling of cell biological systems". *BMC Systems Biology* 5 (166).

Merali, Z. 2010, October. "Computational science: ...Error". *Nature* (467): 775–777.

Miller, J., J. Han, and M. Hybinette. 2010. "Using Domain Specific Language for modeling and simulation: ScalaTion as a case study". In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan. Piscataway, New Jersey: IEEE.

Pawlikowski, K., H. Jeong, and J. Lee. 2002. "On credibility of simulation studies of telecommunication networks". *IEEE Communications Magazine* 40 (1): 132–139.

Peng, D., R. Ewald, and A. M. Uhrmacher. 2014. "Towards Semantic Model Composition via Experiments". In *Conf. on Principles of Advanced and Distributed Simulation (PADS)*: ACM.

Perrone, L. F., T. R. Henderson, Felizardo, V. D., and M. J. Watrous. 2013. "The Design of an Output Data Collection Framework for ns-3". In *Proceedings of the 2013 Winter Simulation Conference*, edited by R. Pasupathy, S.-H.Kim, A. Tolk, R. Hill, and M. Kuhl. Piscataway, New Jersey: IEEE.

Perrone, L. F., C. S. Main, and B. C. Ward. 2012. "SAFE: Simulation Automation Framework for Experiments". In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. Uhrmacher. Piscataway, New Jersey: IEEE.

Prill, R. J., P. A. Iglesias, and A. Levchenko. 2005, October. "Dynamic Properties of Network Motifs Contribute to Biological Network Organization". *PLOS Biology* 3 (11): e343.

Rahmandad, H., and J. D. Sterman. 2012. "Reporting guidelines for simulation-based research in social sciences". *System Dynamics Review* 28 (4): 396–411.

Reiter, M., U. Breitenbucher, O. Kopp, and D. Karastoyanova. 2012. "Quality of data driven simulation workflows". In *Conf. on E-Science (e-Science)*: IEEE.

Ribault, J., O. Dalle, D. Conan, and S. Leriche. 2010. "OSIF: A Framework to Instrument, Validate, and Analyze Simulations". In *Proc. of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*: ICST.

Riley, G., and T. Henderson. 2013. *Modeling and Tools for Network Simulation*, Chapter The ns-3 Network Simulator, 15–34. Springer.

Rizk, A., G. Batt, F. Fages, and S. Soliman. 2009. "A general computational method for robustness analysis with applications to synthetic gene networks". *Bioinformatics* 25 (12): i169–i178.

Rybacki, S., F. Haack, K. Wolf, and A. M. Uhrmacher. 2014. "Developing simulation models - from conceptual to executable model and back - an artifact-based workflow approach". In *Proc. of the 7th International ICST Conference on Simulation Tools and Techniques (SIMUTools 2014)*: ICST.

Rybacki, S., J. Himmelspach, and A. M. Uhrmacher. 2012. "Using Workflows to Control the Experiment Execution in Modeling and Simulation Software". In *Proc. of the 5th International ICST Conference on Simulation Tools and Techniques (SIMUTools 2012)*, 93–102: ICST.

Sargent, R. G. 2013. "Verification and validation of simulation models". *Journal of Simulation* 7 (1): 12–24.

Tomita, T., S. Hagihara, and N. Yonezaki. 2011. "A Probabilistic Temporal Logic with Frequency Operators and Its Model Checking". In *Proc. of the 13th International Workshop on Verification of Infinite-State Systems, Taipei, Taiwan, 10th October 2011*, edited by F. Yu and C. Wang, Volume 73 of *Electronic Proceedings in Theoretical Computer Science*, 79–93: Open Publishing Association.

van Deursen, A., P. Klint, and J. Visser. 2000, June. "Domain-specific languages: an annotated bibliography". *SIGPLAN Notices* 35 (6): 26–36.

von Laszewski, G., M. Hategan, and D. Kodeboyina. 2006. *Workflows for e-Science*, Chapter Java CoG Kit Workflow, 320–339. Springer.

W3C 1999. "XML Path Language (XPath) Version 1.0". http://www.w3.org/TR/xpath/.

Waltemath, D., R. Adams, F. Bergmann, M. Hucka, F. Kolpakov, A. Miller, I. Moraru, D. Nickerson, S. Sahle, J. Snoep, and N. Le Novere. 2011. "Reproducible computational biology experiments with SED-ML - The Simulation Experiment Description Markup Language". *BMC Systems Biology* 5:198.

Weidlich, M., G. Decker, A. Großkopf, and M. Weske. 2008. "BPEL to BPMN: The Myth of a Straight-Forward Mapping". In *On the Move to Meaningful Internet Systems: OTM 2008*, edited by R. Meersman and Z. Tari, Volume 5331 of *Lecture Notes in Computer Science*, 265–282. Berlin, Heidelberg: Springer.

Wood, P. T. 2012, April. "Query Languages for Graph Databases". *SIGMOD Rec.* 41 (1): 50–60.

Zeigler, B. P. 1984. *Multifacetted Modelling and Discrete Event Simulation*. 1st ed. Academic Press.

AUTHOR BIOGRAPHIES

JOHANNES SCHÜTZEL and **DANHUA PENG** pursue Ph.D. degrees in the Modeling and Simulation Group at the University of Rostock. Their email addresses are johannes.schuetzel@uni-rostock.de and danhua.peng2@uni-rostock.de.

ADELINDE UHRMACHER is Professor at the Institute of Computer Science and head of the Modeling and Simulation Group at the University of Rostock. Her email address is adelinde.uhrmacher@uni-rostock.de.

L. FELIPE PERRONE is Associate Professor of Computer Science at Bucknell University. His email address is perrone@bucknell.edu.