Using AI Planning to Automate the Performance Analysis of Simulators

Roland Ewald
University of Rostock
A.-Einstein Str. 22
Rostock, Germany
roland.ewald@uni-rostock.de

ABSTRACT

Analyzing simulation algorithm performance is cumbersome: execute some runs, observe a performance metric, and analyze the results. Often, the results motivate follow-up experiments, which in turn may lead to additional experiments, and so on. This time-consuming and error-prone process can be automated with planning approaches from artificial intelligence, making simulator performance analysis more convenient and rigorous. This paper introduces ALESIA, a prototypical system for automatic simulator performance analysis. It is independent of any specific simulation system and realizes a hypothesis-driven approach to evaluate performance.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems; I.2.8 [**Artificial Intelligence**]: Plan execution, formation, and generation

General Terms

Experimentation, Performance

Keywords

Performance Analysis, Experiments, Planning, Simulation

1. INTRODUCTION

Many new simulation algorithms are presented every year, e.g., to address new trends in hardware (like GPUs or many-core CPUs), new methodologies (like multi-scale or approximative simulation), or new requirements (like scalability or adaptivity). Yet, the methodology with which these simulators are evaluated has not been improved much over the past decades. Performance experiments are still done manually, in the sense that experiment setup, experiment execution, data analysis, and potential follow-up experiments are typically configured and triggered by hand.

Published in proceedings of the SIMUTools 2014 (see http://simutools.org)

However, simulation algorithm performance studies are often quite similar. They investigate similar metrics — usually execution time — and have to cope with similar challenges: the benchmark models must be realistic, comparisons to alternative configurations or competing approaches must be statistically sound, execution times may vary drastically between models, and so on. With the advent of general simulation frameworks (e.g., [22, 12]) and abstraction layers for simulation experimentation (e.g., [21, 7]), it is possible to further automate simulator performance analysis, making it easier and faster to carry out.

Additionally, automated experimentation tools could also reduce some forms of cognitive bias introduced by the experimenters. Bias comes in many forms (e.g., see [25]) and may lead, for example, to wrong conclusions regarding the generality of results. Note, however, that such tools may also introduce bias themselves, e.g., by choosing one kind of experimentation technique over another. In any case, automated experimentation tools could help to ensure that experimentation techniques are used as intended, and thereby safeguard the validity of a performance study.

The main idea put forward in this paper is to further automate simulator performance analysis by closing the gap between analyzing results and starting follow-up experiments. This is done by employing a planning algorithm that triggers configurable sub-experiments. Ideally, an experimenter would only have to model the domain of interest (e.g., simulation algorithms and benchmark models) and could then submit hypotheses to the system. The goal of the planner is to falsify a given hypothesis by executing sequences of suitable sub-experiments. If the hypothesis cannot be falsified by these attempts, it is corroborated.

After illustrating the purpose of our approach with a more detailed usage scenario (Section 2), we sketch out our current prototype (Section 3) and give a brief example (Section 4). Then, we discuss related approaches (Section 5) and future work (Section 6).

2. USAGE SCENARIO

Consider the comparison of some simulator A with a competing simulator B. Simulator B is optimized for large models, so we assume it is faster than A if the model size exceeds x_{hyp} entities. We can formally express this hypothesis with first-order predicate calculus, e.g.,

$$\forall m \in \mathcal{M} : moreEntities(m, x_{hyp}) \Rightarrow faster(B, A, m) \quad (1)$$

where $\mathcal{M}$ is the set of all models that A and B can sim-

ulate. The predicate $moreEntities$ is true iff the model size exceeds our threshold x_{hyp}, and the predicate $faster$ is true iff simulator B is faster than A when simulating m. Note that $faster$ is also just a predicate and can be defined arbitrarily, e.g., to compare both wall-clock times and CPU times: $faster(A,B,m) \iff faster_{CPU}(A,B,m) \wedge faster_{WCT}(A,B,m)$.

To falsify the hypothesis in (1), we have to find a counterexample, i.e., a model $m' \in \mathcal{M}$ that contains more than x_{hyp} entities, so that $moreEntities(m', x_{hyp})$ is true, but where simulator A outperforms simulator B, so that $faster(B,A,m')$ is false. $\mathcal{M}$ is the set of all possible input models, so it is prohibitively large in general. We therefore sample and test only a subset of elements from $\mathcal{M}$.

Given some model $m \in \mathcal{M}$, we can check whether the implication in (1) is true by evaluating both $moreEntities$ and $faster$. While $moreEntities$ merely requires to compare the number of model entities with x_{hyp}, $faster$ requires to execute simulations of model m with both simulator A and simulator B. Execution times are noisy, so multiple simulation runs might be required to arrive at a conclusion, and a statistical test should be used to interpret the results. The following outcomes are possible:

1. B outperforms A: $faster(B,A,m) \wedge \neg faster(A,B,m)$
2. A outperforms B: $faster(A,B,m) \wedge \neg faster(B,A,m)$
3. A and B perform similarly: $\neg faster(A,B,m) \wedge \neg faster(B,A,m)$
4. A, B, or both crashed.
5. Another error occurred.

While a failure handling mechanism can deal with outcomes 4 and 5, the relevance of outcomes 1–3 depends on the goals of the experiment, i.e., the hypothesis to be falsified. In case of the hypothesis in (1), encountering outcomes 2 or 3 should terminate the experiment, and the experimenter should be notified about the counterexample m. If this outcome could not be provoked for any model until the allocated computational resources are exhausted, the hypothesis could not be falsified and is thus corroborated.

Choosing a suitable experiment setup.

Although the hypothesis in (1) is rather simple, there are various ways to attempt its falsification. The most straightforward approach is to randomly sample from $\mathcal{M}$, execute both simulators A and B on m for a fixed number of times, and then apply a statistical test to their run times. Yet, it is still not clear how large the fixed number of executions should be. Assuming a fixed budget of computing time, we can either test fewer models more rigorously, or more models less rigorously.

Also, the measured execution times might be normally distributed. If this is the case, we could replace our (non-parametric) statistical test with a test that assumes normality and is thus more powerful. By letting a more suitable statistical test decide about additional simulation runs, we could hence save many unnecessary simulation runs and thus explore more models from $\mathcal{M}$ instead — our experiment setup would be more efficient.

Another approach would be to train performance estimators for the simulators A and B, in order to predict their execution times with respect to various model features. This could help to identify the regions of the model space $\mathcal{M}$ where B may not outperform A, and to focus the sampling on these regions. However, this approach is only promising if the performance of A and B is indeed predictable from the observed model features.

Yet another approach would be to first analyze the sensitivity of simulator performance with respect to model size. The resulting knowledge on interactions between different factors could also help to sample more promising candidates from $\mathcal{M}$, even if simulator execution times are hard to predict.

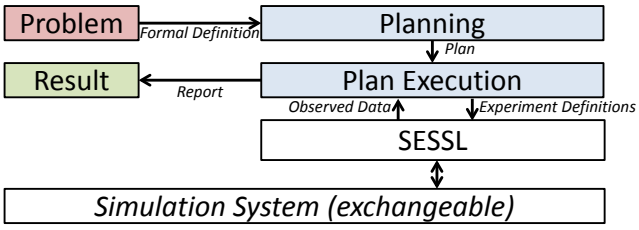

Figure 1: Execution of Alesia: a problem (red) is solved by planning and executing experiments (blue), which yields a result (green).

Potential for Automation.

Each of the above experiment setups would work in principle. The list is not exhaustive, and there may be many more alternatives for more complex hypotheses. However, more powerful experiment setups are also more difficult to realize, as they often require additional methods from machine learning, operations research, or statistics. Moreover, many of these techniques are only applicable under certain circumstances, so that their preconditions (e.g., assumptions regarding statistical properties) need to be checked beforehand. The suitability of a setup also depends on user preferences, available performance knowledge (e.g., from prior experiments or theoretical analysis), and the available techniques, as well as their interdependencies. For example, a non-uniform sampling from $\mathcal{M}$ may require the results of a prior sensitivity analysis.

All these techniques, their use cases, and their preconditions are hard to manage manually. Therefore, we need dedicated software systems to automate performance analysis. They can assist experimenters by planning and executing suitable experiment sequences automatically, thereby exploiting all available techniques and domain-specific knowledge. Ideally, an experimenter would only have to declare which simulators are of interest, which benchmark models are available (i.e., how to sample from $\mathcal{M}$), and which hypothesis should hold. By supporting more general hypotheses, e.g., referring to the predictability of simulator performance, one could also conduct exploratory experiments.

3. A SYSTEM FOR AUTOMATIC SIMULATOR PERFORMANCE ANALYSIS

Alesia, our research prototype for automatic simulator performance analysis, is written in Scala [20]. Thus, it is interoperable with any software that runs on the Java virtual machine (JVM). To separate Alesia from the simulation system under study, all experiment executions are delegated

```
1 val result = submit (
2   SingleModel("java://examples.sr.LinearChainSystem")
3 ) (
4   TerminateWhen(WallClockTimeMaximum(seconds = 30))
5 ) (
6   exists >> model | hasProperty("qss")
7 )
```

Figure 2: Sample problem definition, describing the application domain (line 2), user preferences (l. 4), and the hypothesis to be falsified (l. 6).

```
Action loadSingleModel:
 precondition: ¬depleted ∧ ¬loadedModel
 effect: depleted ∨ loadedModel //'or': non-determinism

Action checkQSSProperty:
 precondition: loadedModel
 effect: hasProperty(qss)∨ //'or': non-determinism
                (¬hasProperty(qss) ∧ ¬loadedModel)
```

Figure 3: Actions for the problem from Figure 2.

to a general interface. We provide a default implementation of this interface that is based on SESSL [7], a domain-specific language (DSL) to set up simulation experiments in Scala. SESSL already supports several simulation systems, including one that does not run on the JVM. Integrating a simulation system supported by SESSL into Alesia is trivial: for example, the integration of the modeling and simulation framework JAMES II [12], which is used in the example of Section 4, consists of three lines of code. Figure 1 illustrates the overall structure of Alesia. In the following, we will walk through the major features of the system, in their order of execution.

3.1 Problem Definition

To start an experiment, the user has to specify a hypothesis similar to (1). In principle, experiment hypotheses can only be falsified (e.g., see [24]), yet our system currently interprets hypotheses as goals. Thus, to falsify a hypothesis h, the system needs to be configured with a hypothesis $\neg h$. Since the experiment's hypothesis refers to entities in some application domain, e.g., certain simulators and benchmark models, all relevant elements of that domain need to be specified as well. This specification may also include existing knowledge, e.g., stemming from previous experiments.

Besides specifying application domain and hypothesis, an experimenter may also want to express certain preferences regarding the actual execution. For example, the maximal number of experiment actions to be executed could be given, or the maximum amount of wall-clock time permitted to attempt a falsification.

To make all this as simple as possible, we are developing embedded domain-specific languages for these tasks, i.e., small sub-languages implemented with Scala constructs. Figure 2 shows a very simple problem definition using these languages. It declares the existence of a benchmark model (see Section 4), it restricts overall execution duration to a maximum of 30 seconds, and it hypothesizes that there is a model with some property called `qss` (see Section 4). Technically, the experimenter simply calls a method `submit` (line 1 in Figure 2) with three sets of arguments: elements of the application domain, user preferences, and experiment hypothesis. This triggers the execution process and eventually returns the experiment results.

3.2 Planning

3.2.1 Planning Preparation

Not all planning algorithms can deal with all user-defined aspects. For example, resource consumption may be easier to restrict during execution, instead of including it in the formal planning problem. A dedicated component, the *planning preparator*, therefore splits a submitted problem definition into a formal *planning problem* and an *execution context*. The former is processed by the planning algorithm (see Section 3.2.2), while the latter holds all runtime information not required by the planner (see Section 3.3). The planning preparator also performs validity checks on the problem definition. This helps to rule out potential error sources before a user is confronted with a failed planning attempt, which can be hard to understand.

To prepare the planning problem, the planning preparator scans the classpath for *action specifications*. Action specifications implement a common interface, and each may declare arbitrarily many *concrete actions*, depending on the problem definition and on actions declared by other action specifications. In our case, each declared action refers to the execution of a concrete sub-experiment or data analysis task. As the results of such tasks are typically unknown beforehand (see Section 2), the action effects are defined to be non-deterministic, as shown in Figure 3.

The declaration of new actions is repeated until no new actions are declared by any specification. This simple setups allows to provide additional action specifications as external plugins. It also allows to resolve inter-plugin dependencies in a decentralized manner, since a specification may be configured to declare no concrete action unless, for example, a certain kind of model is part of the problem definition. This means that action specifications are able to self-select for a problem definition, which reduces the number of concrete actions and thereby the complexity of the planning problem.

Finally, all concrete actions are combined with formal representations of a *start state* and *goal states*; together they define the planning problem. The start state is derived from the application domain (line 2 in Figure 2), the goal states are derived from the hypothesis (line 6 in Figure 2).

3.2.2 Non-Deterministic Planning

Since the exact outcome of an action, i.e., a *sub-experiment*, is often unknown beforehand (see Section 3.2.1), we have to solve a non-deterministic planning problem. Formally, this means that the planning domain is a non-deterministic state automaton, so there may be multiple potential successor states per state transition. Here, state transitions correspond to actions, so that each successor state represents a possible outcome of a sub-experiment.

Non-deterministic planning is a difficult problem, as the uncertainty regarding the outcome of each action must be accounted for. This uncertainty also reflects on the kinds of result a planning algorithm may yield (e.g., see [8, p. 403 et sqq.]): weak plans *may* lead to a goal state, while strong plans *will* lead to a goal state, regardless of the spe-

```
if(¬depleted ∧ ¬loadedModel)
    use loadSingleModel
else if(loadedModel)
    use checkQSSProperty
else error
```

Figure 4: A policy for the problem from Figure 2, relying on the actions from Figure 3.

cific outcomes of any action. So-called *strong-cyclic* plans are in-between: they lead to a goal state under the assumption that there are no infinite cycles in the state space. It seems unrealistic to expect strong plans in a domain like performance analysis: if we could be sure that a hypothesis can be falsified, we would not have to carry out the experiments in the first place. Therefore, experiment plans will typically be either weak or strong-cyclic.

To create weak, strong, or strong-cyclic plans, ALESIA currently uses symbolic model checking [8, p. 403 et sqq.], following the approach by Cimatti et al. [4] (see Section 5 for a discussion on alternatives). The main idea is to represent *sets* of states via boolean functions, which can be compactly represented as *boolean decision diagrams* (BDDs, e.g., see [17]).

Start state,[1] goal states, action preconditions, and action effects are all represented by BDDs. This allows to express operations on sets of states as logical operations on BDDs, e.g., using $f \wedge g$ to intersect the sets of states where f is true and g is true, respectively. On this basis, one can now define planning algorithms that find weak, strong, or strong-cyclic plans [4, 8]. For weak and strong planning, we implemented the algorithms as described in [8]. For strong-cyclic planning, we implemented Rintanen's simplified version [23] of an algorithm by Cimatti et al. [4].

While deterministic plans can be represented by action sequences, non-deterministic plans are represented by *policies*, i.e., functions that map elements of the state space to actions. A policy determines which action to choose in a given state, thus allowing, for example, to repeat sub-experiments that failed before. A sample policy is depicted in Figure 4.

3.3 Plan Execution

The generated plan (i.e., policy) is now handed over to an *executor*, which carries out the actual experiment. The executor manages both the current state of the planning domain and the execution context, which holds all other relevant data. At first, the executor queries the policy, which decides upon the actions to be executed in the current state. If multiple actions are available to approach the goal, a custom tie-breaking component is called to pick one of them. This allows to implement additional heuristics, for example, to detect when an action is executed repeatedly without affecting the current state, and to then choose an alternative action. Such heuristics are particularly important if only a weak plan was found (see Section 3.2.2).

Then, the selected action is executed and its results are used to update both the current state of the planning domain and the execution context. To make the results of the action accessible to other actions, the execution context stores associations between literals in the planning domain and result

[1]Technically, a function defining a *set* of start states.

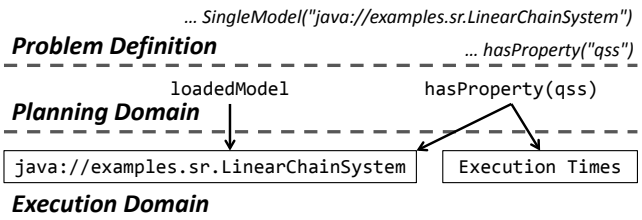

Figure 5: Alesia separates problem definition, planning domain, and execution domain. Arrows show the links between literals of the planning domain and additional data required during execution.

data. For example, the existence of a performance estimator for some simulator A might be represented by a literal $PerfEstimator_A$ in the planning domain, which would be associated with the actual estimator object stored in the execution context. A literal can be associated with multiple objects in the execution context, and multiple literals can be associated with the same object. This allows to share data between actions: e.g., multiple analysis actions could rely on the same performance data. The separation of planning domain and execution domain is sketched in Figure 5.

Failures occurring during action execution are handled by a dedicated component, which may decide, for example, to abort the experiment or to ignore the error. The desired behavior can be configured by the experimenter, via user preferences (see Section 3.1). A separation of failure handling and planning allows us to reduce the degree of uncertainty, since in principle any kind of action can fail (see example in Section 2). For experiments regarding robustness, actions testing a simulator may catch potential exceptions themselves, and then return normally. Thus, the implementor of an action is free to decide what is considered a failure, and what can be expected as a normal result of the experiment.

Our current implementation simply executes the above procedure sequentially, one action at a time. It should also be possible to execute multiple available actions in parallel, since sub-experiments are independent of each other.[2] Thus, we plan to distribute suitable sub-experiments to remote machines with an identical hard- and software setup. More elaborate executor implementations could also exploit heterogeneous sets of machines, thereby improving the applicability of the results.

3.4 Report Generation

Since the increased degree of automation may hide problems with the experiment setup, a crucial feature is the generation of detailed result reports. These should include the raw data collected during the experiment, so that it is still available for manual analysis. Besides increasing the trust of the experimenters, this also helps with diagnosing experiment failures and with defining follow-up hypotheses. Currently, our system generates a result report that describes each iteration of the plan executor (see Section 3.3), i.e., the current state of the planning domain, which experiment action was chosen, and how the state changed after its execution. In the future, we plan to integrate plots for the

[2]However, this would imply, for example, that the effects of executed actions must not cancel each other out. Such properties can be checked at runtime.

numerical results of each action, which will further help experimenters with their analysis.

4. EXAMPLE: STOCHASTIC SIMULATION ALGORITHMS

We now consider a more concrete example, similar to the scenario described in Section 2. Our goal is to analyze two variants of a *stochastic simulation algorithm* (SSA) [11], as implemented in the modeling and simulation framework JAMES II [12]: the *Direct Method* (DM) [10] and the *Next Reaction Method* (NRM) [9]. We assume that there is a parameterization of the *Linear Chain System* benchmark model [3] (a model consisting of a chain of chemical reactions) where DM should be faster than NRM, in terms of wall-clock time (WCT). Additionally, we demand that this model instance should be in a 'quasi-steady state' (`qss`). Here, a 'quasi-steady state' means that performance-relevant model features (e.g., model structure, number of entities, etc.) are relatively stable during a simulation run, so that execution time is approximately proportional to the number of executed simulation events (see [6]).

The above scenario can be declared in ALESIA as shown in Figure 6. The user domain contains a set of model instances (lines 2–4), defined by a model URI (line 2) and two model parameters (lines 3 and 4). Additionally, the user domain contains the declaration of two simulation algorithms (lines 5 and 6), which define unique string IDs (`nrm`, `dm`) for the SESSL entities (see Section 3) that represent the simulators. The hypothesis (line 9) states that there is a model with a certain property (here, `qss`), and that DM is faster than NRM on this model. Additionally, we define that each simulator should be executed for about one second of wall-clock time (line 7) and that no more than 100 experiment actions shall be executed (line 8).

Now, the overall procedure as outlined in Section 3.2 begins. At first, the planning preparator (see Section 3.2.1) queries all action specifications. They declare an action for sampling a model instance from the model set, an action for checking the `qss` property of a given model, and an action to compare two simulators on a given model. To make a valid comparison, however, one also needs to know for how many simulation events the given model should be simulated until execution times are not dominated by stochastic noise or warm-up cost anymore. This data can be provided, for example, by a 'calibration' action implementing an approach as described in [6], and which is thus declared here as well. If there would be no such action (or any alternative), the user would be notified that no plan could be found.

In our case, the planner (see Section 3.2.2) is able to find a (strong-cyclic) execution policy and hands it over to the executor (see Section 3.3). The executor starts with sampling a model instance from the model set. Then, it could either calibrate the simulators to this instance and compare them, or check its `qss` property — the order of experiments is not fixed, but depends on action interdependencies, e.g., the calibration has to precede the comparison. The simulator comparison applies a statistical test to the observed execution times. If the performance comparison is 'successful', i.e., DM outperforms NRM, and the `qss` property holds for the given model, plan execution stops. Otherwise, the action executions lead to a removal of the literals that represent the current model instance (and its calibration data)

```
1 val result = submit(
2   ModelSet("java://examples.sr.LinearChainSystem",
3    ModelParameter("numOfSpecies",1,10,1000),
4    ModelParameter("numOfInitialParticles",10,100,10000)),
5   SingleSimulator("nrm", NextReactionMethod()),
6   SingleSimulator("dm", DirectMethod())
7 ) (DesiredSingleExecutionWallClockTime(seconds = 1),
8  TerminateWhen(MaxOverallNumberOfActions(100))
9 ) (exists >> model | (hasProperty("qss") and
       isFasterWCT("dm", "nrm", model)) )
```

Figure 6: Example scenario.

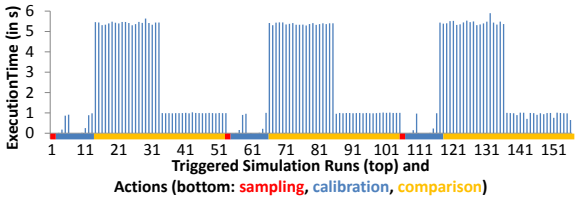

Figure 7: The first actions and corresponding simulation runs executed for the example in Figure 6.

in the planning domain. This causes the policy to again select the model-sampling action, starting a new iteration. Figure 7 shows this for the first nine actions and their corresponding simulation runs (if any), as executed by ALESIA for the given example. The generated report currently documents the sequence of actions and planning domain states.

Note that, while this example is still quite simple, it already involves a random sampling of model instances, simulator calibration (see above), and statistical testing. In the future, we plan to support additional model properties (e.g., stiffness), hypotheses that refer to (multi-dimensional) parameter spaces, sophisticated sampling (e.g., w.r.t. model structure), *sets* of simulator configurations, and advanced analysis methods (as discussed in the next section).

5. RELATED WORK

In contrast to approaches for defining and executing simulation experiments, such as SAFE [21] or SESSL [7], ALESIA shall automate experiment design and result analysis. Thus, it can be used on top of these systems, but serves another purpose and is restricted to simulator performance analysis (at least for now).

Both the empirical analysis of algorithm performance and the application of AI methods to automate experimentation, particularly in the life sciences [19], are areas of active research. Experimental algorithmics [18], i.e., the empirical evaluation of algorithm performance, may guide software development [14] and may also lead to new theoretical insights [13]. Many techniques developed in this context are of particular interest to ALESIA, as they correspond to experiment actions in our prototype. For example, approaches to automatically configure parameterizable algorithms, such as F-Race [2] or ParamILS [16], can be used to find the best configurations of two simulators before comparing them. Other methods, e.g., to find the most relevant features for performance prediction [15] or to generate portfolios of simulation algorithms [5], will be useful as well.

To the best of our knowledge, AI planning has not yet been

considered to automate the performance analysis of simulation algorithms, nor are there dedicated software systems that integrate techniques for their empirical analysis. Our basic planning approach is currently quite similar to planners like MBP [1], but less efficient and less sophisticated. Non-deterministic planning problems can also be solved by representing them as Markov decision processes (e.g., [8, p. 379 et sqq.]) or as satisfiability problems (e.g., [8, p. 437 et sqq.]). Such approaches will be considered to extend the capabilities of ALESIA in the future.

6. CONCLUSIONS & OUTLOOK

This paper introduces ALESIA, a system to automate the performance analysis of simulation algorithms. ALESIA is work in progress, so there are many open issues. For example, the expressiveness of the problem definition DSLs is rather limited so far, and literals of the planning domain are still represented by strings, which is error-prone. In the next months, we plan to integrate our approaches to incremental performance analysis into the system, e.g., to support the analysis of individual simulator components [26]. Other topics of ongoing research are the integration of existing approaches for automatic simulator configuration (see Section 5) and the support for distributed execution. ALESIA is open source; the repository with the current prototype can be found at `https://bitbucket.org/alesia/alesia-core`.

Acknowledgments

This research was supported by the German research foundation, via research grant EW 127/1-1 (ALESIA). I would like to thank the reviewers for their helpful comments.

7. REFERENCES

[1] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. MBP: a model based planner. In *Proceedings of the IJCAI'01*, 2001.

[2] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and iterated F-Race: An overview. In *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, 2010.

[3] Y. Cao, H. Li, and L. Petzold. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *J. Chem. Phys.*, 121(9):4059–4067, 2004.

[4] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, July 2003.

[5] R. Ewald, R. Schulz, and A. M. Uhrmacher. Selecting simulation algorithm portfolios by genetic algorithms. In *Workshop on Principles of Advanced and Distributed Simulation*, pages 48–56. IEEE CPS, 2010.

[6] R. Ewald and A. M. Uhrmacher. Automating the runtime performance evaluation of simulation algorithms. In *Proceedings of the Winter Simulation Conference*, pages 1079–1091. IEEE CS, 2009.

[7] R. Ewald and A. M. Uhrmacher. Setting up simulation experiments with SESSL, 2012. Winter Simulation Conference, Poster.

[8] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann, 2004.

[9] M. A. Gibson and J. Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *J. Chem. Physics*, 104:1876–1889, 2000.

[10] D. T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. Comput. Phys.*, 22, 1976.

[11] D. T. Gillespie. Exact Stochastic Simulation of Coupled Chemical Reactions. *Journal of Physical Chemistry*, 81(25), 1977.

[12] J. Himmelspach and A. M. Uhrmacher. Plug'n simulate. In *Proc. of the 40th Annual Simulation Symposium*, pages 137–143. IEEE CS, 2007.

[13] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42(2):201–212, 1994.

[14] H. H. Hoos. Programming by optimization. *Comm. of the ACM*, 55(2):70–80, Feb. 2012.

[15] F. Hutter, H. Hoos, and K. Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In *Learning and Intelligent OptimizatioN Conference*, Jan. 2013.

[16] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[17] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, Mar. 2009.

[18] C. McGeoch. Experimental algorithmics. *Comm. of the ACM*, 50(11):27–31, Nov. 2007.

[19] S. H. Muggleton. 2020 computing: Exceeding human limits. *Nature*, 440(7083):409–410, Mar. 2006.

[20] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2nd edition, Jan. 2011.

[21] L. F. Perrone, C. S. Main, and B. C. Ward. SAFE: simulation automation framework for experiments. In *Proceedings of the Winter Simulation Conference*. WSC, 2012.

[22] J. Ribault, F. Peix, J. Monteiro, and O. Dalle. OSA: an integration platform for component-based simulation. In *Proc. of the 2nd Int'l Conference on Simulation Tools and Techniques (SIMUTools)*. ICST, 2009.

[23] J. Rintanen. Complexity of conditional planning under partial observability and infinite executions. In *Proceedings of the 20th European Conference on Artificial Intelligence*, pages 678–683. IOS Press, 2012.

[24] H. Sankey. Scientific method. In S. Psillos and M. Curd, editors, *The Routledge Companion to Philosophy of Science*, chapter 9, pages 248–258. Taylor & Francis, Jan. 2008.

[25] A. Tversky and D. Kahneman. Judgment under uncertainty: Heuristics and biases. *Science*, 185(4157):1124–1131, Sept. 1974.

[26] J. Wienß, M. Stein, and R. Ewald. Evaluating simulation software components with player rating systems. In *Proc. of the 6th Int'l Conference on Simulation Tools and Techniques (SIMUTools)*. ICST, 2013.