Noname manuscript No.
(will be inserted by the editor)

ExpRunA: a domain-specific approach for technology-oriented experiments

Eneias Silva · Alessandro Leite · Vander Alves · Sven Apel

Received: date / Accepted: date

Abstract Conducting technology-oriented experiments (i.e., experiments in which treatments are applied to objects by a computer-based tool) without proper tool support is often a time-consuming and highly error-prone task. Although many techniques have been proposed to help conducting controlled experiments, none of them simultaneously addresses (1) the executable specification of experiments at a high level of abstraction; (2) automated treatment execution and automated data analysis from the experiment specification; and (3) formal guaranties of the correctness of results according to an experiment specification for technology-oriented experiments. To address these issues, we provide a Domain-Specific Modeling approach to create a Web-based tool (*ExpRunA*) comprising a Domain-Specific Language named *ToExpDSL*, execution and analysis script generators, a supporting framework, and a running infrastructure. An experimenter uses *ToExpDSL* to specify an experiment using experimentation concepts. From this specification, applications corresponding to the underlying treatments are executed, execution results are collected and analyzed, and, finally, the analysis results are presented to the experimenter. We establish the consistency of such results with respect to the experiment specification by formalizing and proving key correctness properties of *ExpRunA*. We empirically evaluated *ExpRunA* with respect to automation by replicating three already published experiments; we evaluated the level of abstraction by a qualitative assessment. Our empirical evaluation shows that *ToExpDSL* is expressive enough to specify three technology-oriented experiments and that *ExpRunA* can be used to enable sound automation of execution and analysis from the specification of technology-oriented experiments at a high level of abstraction.

Keywords Controlled experiments · Technology-oriented experiments · Domain-specific modeling · Domain-specific language

E. Silva (✉) · A. Leite · V. Alves
Department of Computer Science
C.P. 4466, University of Brasília, 70910-900, Brasília, Brazil
Tel.: +55 61 3107-3675
E-mail: eneiascs@gmail.com

Sven Apel
Saarland Informatics Campus, Saarland University.
Campus E1 1, 66123 Saarbruecken, Germany

1 Introduction

Experimentation includes the empirical investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables [18]. Independent variables are all variables in a process that are manipulated and controlled, whereas the dependent variable is the outcome of an experiment that we want to study to see the effect of the changes in some input [53, 26]. In software engineering, for instance, experimentation enables researchers contrasting suppositions, assumptions, speculations, and beliefs that abound in software construction with facts [26], which is essential for the rigorous development of the field and ultimately its relevance. Experiments can be *human-oriented* or *technology-oriented* [53]. In the former, a person applies treatments to experimental objects, whereas, in the latter, treatments are applied to experimental objects by a computer-based tool. A treatment is one particular value of an independent variable, which is any characteristic that is intentionally varied during experimentation to examine its influence on the dependent variable. The

experimental objects are the objects on which the experiment is run [53, 26].

Technology-oriented experiments are important not only in software engineering but also in other research fields, such as bioinformatics, engineering, physics, and chemistry [14, 45, 38, 24]. Technology-oriented experiments can be used in several ways. First, software can be used to evaluate methodologies or approaches [36]. Moreover, software can be used in simulation-based studies [6]. For example, in engineering, systems can be simulated by using software to avoid the costs of building real systems [45]. In some studies, (e.g., use and occupation of the soil), evaluations cannot be performed in the wild, so they need to rely on simulations [39]. Furthermore, software can also be used together with physical instruments. For example, chemistry and physics laboratories can have instruments connected to computers to automate experiments [24, 38]. In this work, we focus on such technology-oriented experiments, and hereafter we use the term *experiment* to refer to this context, which includes not only *in silico* simulation based studies [47] but also experiments that evaluate algorithms or computer-based tools. For example, Lanna et al. [30] presented a novel *feature-family-based* analysis strategy to compute the reliability of all products of a software product line. To evaluate their strategy, the authors created a tool named ReAna[1] and used it to compare the performance of different reliability analysis strategies for software product lines.

Conducting an experiment is often a complex and time-consuming task. Since experimentation involves many steps, such as goal definition, planning, execution, analysis, and packaging, all steps must be performed in a systematic and consistent way to achieve a replicable experiment and valid results [26, 53]. Since the scale of scientific problems has been increasing, this is reflected not only in data size but also on the complexity of the computer-based tools required to investigate such problems [55, 43]. Thus, tools for experimentation must run on an infrastructure that provides computing power, data storage, and network resources. However, deploying and executing applications in such infrastructure (e.g., a cloud computing infrastructure) are complex tasks and require advanced computational skills [28]. Likewise, data analysis requires knowledge on statistics so that results can be correctly analyzed and interpreted. Therefore, conducting controlled experiments is an error-prone task.

There is a number of approaches supporting experimentation, focusing on distinct phases of the experimentation process, and supporting human-oriented or technology-oriented experiments (Section 6). Although these approaches help in conducting controlled experiments, none of them simultaneously addresses the executable specification of experiments at a high level of abstraction; the automated treatment execution and automated data analysis from the experiment specification; and formal guaranties of the correctness of results with respect to the specification of technology-oriented experiments. The lack of proper tool support may lead not only to extra time or resources consumption but also to incorrect results.

To address these issues, we propose a Domain-Specific Modeling (DSM) approach [27] supporting technology-oriented experiments. The approach is implemented as a Web-based tool and comprises a Domain-Specific Language (DSL), named *ToExpDSL*, execution and analysis script generators, a running infrastructure to execute the generated scripts, and a supporting framework integrating the previous components as part of a tool named *ExpRunA*[2]. First, one uses *ToExpDSL* to specify experiments. Next, the aforementioned generators generate execution and analysis scripts from this experiment specification. The running infrastructure executes the execution script, generating experimental data, and then executes the analysis script on these data. Finally, the supporting framework presents the results. The supporting framework integrates all the components and interacts with the running infrastructure to start and monitor execution, and also to analyze the results. The whole procedure of generating execution and analysis scripts, executing, and analyzing an experiment from an experiment specification has been formally specified, and key correctness properties have been stated. A formal proof of these properties assures the correctness of results according to the experiment specification.

We evaluated *ExpRunA* with respect to abstraction, automation, and correctness. To show that *ToExpDSL* raises the level of abstraction of experiment specifications, we evaluated it by an analytical comparison between Domain-Specific Language (DSL) concepts and experimentation concepts, and by comparing the level of abstraction of experiment specifications across different studies. Although the experimenter must learn a new language, the results suggest that the use of *ToExpDSL* raises the level of abstraction such that it matches the experimenters intention and intuition properly. By comparing *ToExpDSL* constructs with domain concepts, we found that 54% are high-level constructs, 15% are mid-level constructs, and 31% are low-level constructs, according to their relation with domain concepts. To demonstrate that *ExpRunA* can automate experiment execution and analysis, we replicated three already pub-

[1] https://github.com/SPLMC/reana-spl/.

[2] https://expruna.github.io/.

lished experiments using it. Our results suggest that *ToExpDSL* is sufficiently expressive to specify technology-oriented experiments and that *ExpRunA* can be used to enable sound automation of execution and analysis from the specification of technology-oriented experiments. We assured correctness by proving key formal properties of the formal specification. In summary, we make the following contributions:

- We present a Domain-Specific Modeling (DSM) approach that supports technology-oriented experiments (Section 3), comprising a DSL (Section 3.2), named *ToExpDSL*, execution and analysis script generators (Sections 3.3 and 3.4), a running infrastructure to execute the generated scripts (Section 3.5), and a supporting framework integrating the previous components (Section 3.6).
- We present a Web-based tool (*ExpRunA*) that implements the Domain-Specific Modeling (DSM) approach (Section 4), providing a means to specify executable experiment specifications at a high level of abstraction, automated execution, data analysis, and results presentation.
- We empirically evaluate the practical applicability of *ExpRunA* to provide automation in the experimentation process and its level of abstraction (Sections 5.1 and 5.2).
- We present a formal model of the whole procedure and proofs of correctness (Section 3).
- We present a supplementary Website[3] where *ExpRunA* is available and with the complete specification, scripts files and results of our evaluation, as well instructions for future replications.

2 Motivation

This section presents and motivates the research problems addressed by *ExpRunA*. We state each problem and illustrate it with examples, providing remarks that highlight its relevance.

Problem 1 An experimenter needs to deal with different levels of abstraction while specifying an experiment and writing execution and analysis scripts. High-level specifications are usually in natural language, which may lead to ambiguity, inconsistency, and lack of information [15] and are not executable. On the other hand, executable specifications are usually written in general purpose languages, at a low level of abstraction.

Example 1 Bak and Duggirala [4] presented a technique to perform simulation-equivalent reachability and safety verification of linear systems with inputs. To evaluate their proposal, they created a tool named Hylaa (HYbrid Linear Automata Analyzer). Their experiment was specified, executed, and measured using Python scripts, [4] an excerpt of which is presented in Listing 1. Overall, the script examines the effects of optimizations for computing reachability for linear time-invariant systems with inputs. Optimizations, which correspond to treatments, are defined in Lines 4–12. In fact, each optimization is defined by appending distinct parameters to the tool (Lines 6 and 10). To measure runtime, each optimization is applied to the input file (`io.xml`). In addition, the number of steps in the problem is varied by changing the step size. Thus, each step size used to run the tool corresponds to an experimental object. The first experimental object is defined by `step_size` variable (Line 17). The following objects are defined in Line 29 inside a loop (Line 18) until the timeout is reached (Line 27). Each treatment is applied to an experimental object in Line 22. This is executed inside a loop (Line 21), which is repeated the number of times defined in the variable `num_trials` (Line 3).

Listing 1: Excerpt of an execution script in Python [4]

```
1  def measure():
2      timeout_secs = 15
3      num_trials = 10
4      tools.append('hylaa')
5      labels.append('Hylaa')
6      tool_params.append('--settings
           settings.print_output=False')
7      input_xml.append('io.xml')
8      tools.append('hylaa')
9      labels.append('Warm')
10     tool_params.append('--settings
           settings.print_output=False ' +
11                        'settings.opt_decompose_lp=False')
12     input_xml.append('io.xml')
13     for i in xrange(len(tools)):
14         tool = tools[i]
15         label = labels[i]
16         with open('out/result_{}.dat'.format(label), 'w') as f:
17             step_size = 0.2
18             while True:
19                 total_secs = 0.0
20                 measured_secs = []
21                 for _ in xrange(num_trials):
22                     res = e.run(print_stdout=True,
                               run_tool=True)
23                     runtime = res['tool_time']
24                     measured_secs.append(runtime)
25                     total_secs += runtime
26                 avg_runtime = total_secs / num_trials
27                 if avg_runtime > timeout_secs:
28                     break
29                 step_size /= 1.3
```

[3] https://expruna.github.io/.

[4] https://bit.ly/2NTCuSe.

Example 2 Lanna et al. [30] used Python scripts to perform an experiment comparing the performance of different reliability analysis strategies for software product lines. An excerpt of the experiment execution script[5] is presented in Listing 2. The loop in Line 3 iterates over treatments (`strategy`) and experimental objects (`spl`). Each treatment is applied to each object (Line 5). This execution is repeated the number of times defined in `number_of_runs` (Line 13).

Listing 2: Excerpt of an execution script in Python [30]

```
1  def run_all_analyses(number_of_runs,in_results):
2      all_stats = []
3      for (spl, strategy), command_line in
           CONFIGURATIONS.iteritems():
4          try:
5              stats = run_analysis(spl, strategy,
                   command_line, number_of_runs)
6              all_stats.append(stats)
7              replay.save(AllStats(all_stats), in_results)
8              test_hypotheses(AllStats(all_stats))
9          except:
10             traceback.print_tb(sys.exc_info()[2],
                   limit=None, file=None)
11     return AllStats(all_stats)
12 def run_analysis(spl, strategy, command_line,
       number_of_runs):
13     data = [_run_for_stats(command_line) for i in
           xrange(number_of_runs)]
14     return CummulativeStats(spl, strategy, data)
```

Remark 1 The previous examples suggest that low-level programming details obfuscate experimentation concepts, which hampers their understanding and future replications of the experiments. Although variable names may help, there is no standard way to define experimentation concepts, such as treatments, experimental objects, and the number of tests to run. For instance, `num_trials` in Listing 1 and `number_of_runs` in Listing 2 were used to represent the same concept. Conversely, a single term can have distinct meanings. For instance, `run_analysis` (Listing 2, Line 5) refers to reliability analysis, which is, in fact, part of the execution phase; analysis may also mean statistical analysis, though. Second, there may be inconsistencies between the experiment specification and its execution script. A treatment or an object could be repeated, resulting in unnecessary executions, or a parameter could be incorrectly assigned to a treatment, resulting in wrong results. For example, parameters assigned to the treatment *Warm* (Line 10, Listing 1) could be incorrectly assigned to the treatment *Hylaa* (Line 6).

Problem 2 An experimenter needs to manually create execution and analysis scripts, which is time-consuming and error-prone.

Example 3 The Gnuplot configuration file presented in Listing 3 is used to plot experiment results from data files, which are then used to draw the conclusions of the experiment. Since this file is *manually* created, it may contain wrong correspondences between treatments and execution results. Additionally, there may be inconsistencies between the execution script presented in Listing 1 and the Gnuplot file presented in Listing 3, e.g., each Line from 2 to 6 relates a title to the corresponding execution results. However, a title could be misassigned to a result file, which would lead to a wrong interpretation and thus incorrect results.

Listing 3: Excerpt of a Gnuplot configuration file [4]

```
1  plot \
2      "out/result_Basic.dat" with linespoints title "Basic" ls 1,
           \
3      "out/result_Warm.dat" with linespoints title "Warm" ls 2
           pi -1, \
4      "out/result_Decomp.dat" with linespoints title "Decomp"
           ls 3, \
5      "out/result_Hylaa.dat" with linespoints title "Hylaa" ls 4
           pi -1, \
6      "out/result_NoInput.dat" with linespoints title "NoInput"
           ls 5, \
```

Example 4 Regarding statistical analysis, Lanna et al. [30] manually created a script,[6] an excerpt of which is shown in Listing 4, to check whether two data samples are significantly different. Accordingly, either a nonparametric Mann–Whitney test or a parametric T test can be applied. The script first checks if the assumptions made by the parametric test are met and then apply the corresponding test. An error in this script, for instance, using `p <= SIGNIFICANCE` instead of `p >= SIGNIFICANCE` in Line 22 would lead to the use of a parametric test when it should not be used (due to failing assumption on data normality), and thus, invalidate the results, which may go unnoticed.

Listing 4: Excerpt of a Python analysis script [30]

```
1  def _compare_samples(sample1, sample2):
2      mean1 = mean(sample1)
3      mean2 = mean(sample2)
4      gain = max(mean1, mean2)/min(mean1, mean2)
5      if not _is_normally_distributed(sample1) or not
           _is_normally_distributed(sample2):
6          normality = "Not all are normal"
7          are_equal, details =
               _non_normal_are_equal(sample1, sample2)
```

[5] https://bit.ly/2CvShEU.

[6] https://bit.ly/2Al2uT7.

```
8       else:
9           normality = "All are normal"
10          are_equal, details = _normal_are_equal(sample1,
                sample2)
11      if not are_equal:
12          result = mean1 - mean2
13      else:
14          result = 0
15      aggregated_details = (normality, details,
16                            {"mean 1": mean1,
17                             "mean 2": mean2,
18                             "gain": str(gain) + "x"})
19      return result, aggregated_details
20  def _is_normally_distributed(sample):
21      w, p = normaltest(sample)
22      return p >= SIGNIFICANCE
23  def _non_normal_are_equal(sample1, sample2):
24      u, p = mannwhitneyu(sample1,
25                          sample2,
26                          use_continuity=False)
27      return p >= SIGNIFICANCE, ("Mann-Whitney", {"U":
            u, "p-value": p})
28  def _normal_are_equal(sample1, sample2):
29      equal_vars = _variances_are_equal(sample1, sample2)
30      are_equal, details = _test_normal_equality(sample1,
            sample2, equal_vars)
31      return are_equal, details
32  def _variances_are_equal(sample1, sample2):
33      stat, p = bartlett(sample1, sample2)
34      return p >= SIGNIFICANCE
35  def _test_normal_equality(sample1, sample2,
        equal_variances):
36      stat, p = ttest_ind(sample1, sample2,
            equal_var=equal_variances)
37      method = "T-test" if equal_variances else "Welch"
38      return p >= SIGNIFICANCE, (method, {"statistic": stat,
            "p-value": p})
```

Remark 2 Regarding execution scripts, Listing 1 and Listing 2 also illustrate Problem 2. Further, there are unexplored commonalities between scripts of distinct experiments, such as treatments, objects, and dependent variables definitions, not to mention the application of treatments to objects and the repetition of executions. This results in the development of similar scripts with duplicated code for distinct experiments, which could be error-prone and time-consuming.

Problem 3 In the context of technology-oriented experiments, there is a lack of formal evidence of the correctness of results in relation to the experiment specification.

Remark 3 With correct results, we mean that the results of the overall experimentation process are consistent with the experiment specification. That is, analysis is evaluating execution results that actually correspond to the hypotheses defined in the experiment specification, using a suitable analysis procedure and the correct parameters. Generating execution and analysis scripts can avoid manual errors; however, there could still be systematic errors in code generators.

Example 5 Execution results could be misassigned to the underlying treatments of the hypotheses, which would happen if we misplaced the `labels` parameter of *Hylaa* (Line 5, Listing 1) and *Warm* (Line 9, Listing 1). Or analysis could misplace the execution results in the analysis test, for example, swapping the order of samples when calling `_compare_samples` (Line 1, Listing 4). Analysis would use an unsuitable analysis function if we wrote (or generated) `p <= SIGNIFICANCE` instead of `p >= SIGNIFICANCE` (Line 30, Listing 4), for instance. Either case would lead to incorrect results. Thus, there is still a lack of formal guaranties of the correctness of results with respect to the experiment specification.

3 Method

Our objective is to provide a solution simultaneously addressing (1) the executable specification of experiments at a high level of abstraction; (2) automated treatment execution and automated data analysis from the experiment specification; and (3) formal guaranties of the correctness of results according to an experiment specification for technology-oriented experiments. To address the aforementioned problems, we present a DSM-based solution, which comprises a DSL (Section 3.2), an experiment execution script generator (Section 3.3), an analysis script generator (Section 3.4), a running infrastructure to execute the generated scripts (Section 3.5), and a supporting framework integrating the previous components (Section 3.6). To assure that the experiment results provided by our model are consistent with the experiment specification, we also provide formal definitions and key correctness properties (Sections 3.3 and 3.4).

3.1 Overview

We present a DSM-based solution as depicted in Figure 1. Initially, we created a DSL (*ToExpDSL*), execution and analysis scripts generators, and a supporting framework. *ToExpDSL* is then used by other researchers to specify an experiment.

In *ToExpDSL*, an experiment comprises a set of research hypotheses, each of which is a statement on the measured effects of treatments. To determine the effect of treatments, a research design defines how to apply them to experimental objects; the effect on dependent variables is measured by the corresponding instrumentation. The resulting data points are analyzed to confirm or refute the hypotheses according to statistical tests corresponding to the type of statement on the research hypotheses.

ToExpDSL allows the researcher to specify an experiment focusing mostly on the domain at hand abstracting from low-level details, this way, addressing Problem 1. Validators check the experiment specification in the DSL for syntactic and type-level consistency. Then, the generator uses this specification to create an execution script, which reflects the design of the experiment and includes all information required to run applications (i.e., computer-based tools) related to the treatments defined in the research hypotheses. The generator also produces an analysis script referring to all statistical tests required to test the hypotheses of the experiment. This frees the researcher from the low-level details of manually creating execution and analysis scripts, this way, addressing Problem 2. Currently, the experimental design relies on a (subset of) Cartesian product to relate treatments and experimental objects, and the research hypotheses are limited to the comparison of two treatments at a time; consequently, the statistical tests performed are limited to T-test and Mann–Whitney, depending on normality of the data.

The framework requests the running infrastructure to execute the experiment execution script producing a series of data points. The framework monitors execution and collects partial results. After execution, the framework uses the running infrastructure to automatically collect and analyze data using the previously generated analysis script to confirm or refute the hypotheses specified in the experiment specification. Automated analysis includes significance testing and generation of measurements and plots from data. This helps researchers in performing descriptive analysis, hypothesis testing, and interpreting the results. It is important to note that all of the results are consistent with the experiment specification, which is guaranteed by formal specification and proof of correctness properties, this way, addressing Problem 3. Finally, an analysis report is presented to the experimenter, which packages and presents results and conclusions of the experiment, as well a laboratory package for future replications.

Although our solution helps in these tasks by providing an analysis report, the generated scripts, and the execution results, the experimenter still has to perform some manual tasks, such as interpreting the results, drawing the conclusions, writing replication instructions, and publishing the laboratory package.

3.2 *ToExpDSL*

Following an action research method [18], we developed the DSM solution inspired by the experimental challenges reported by a colleague in our research group [30]. In this context, we followed an iterative process, including the diagnosis of the problems and reflective learning throughout the development.

ToExpDSL is partially based on *ExpDSL* [20], and extends it with new constructs for technology-oriented experiments; theirs was designed for human-oriented experiments. We choose ExpDSL because it has been already empirically evaluated by experiments and case studies and successfully used in a number of experiments. For instance, the authors modeled several controlled experiments with the objective of analyzing ExpDSL's completeness and expressiveness [21].

The syntax of *ToExpDSL*, containing the main constructs of the language, is presented in Listing 5. We represent types as records, and we write $e.hypotheses$ to access the data stored at field *hypotheses* of a given experiment $e \in E$, for example. In addition, we use overlines to represent lists. For instance, *hypotheses* are represented by type H. So, $\overline{H}$ represents a list of hypotheses. We also assume the existence of primitive types, such as *String*, *PosInt*, and *Float*.

An experiment specification E comprises a list of research hypotheses H, an experimental design D, a list of treatments T, a list of experimental objects O, and a list of dependent variables DV (Line 1). Each research hypothesis compares the values of a dependent variable DV corresponding to the execution of each treatment T (Line 2). The experimental design D comprises the number of runs (i.e., the number of times each treatment is applied to the same object) and a design function (Line 3). The design function defines how treatments are applied to experimental objects. For each treatment, a related command is specified (Line 4). This command represents the command line used to run that treatment in the infrastructure. Likewise, for each experimental object O, an argument is defined (Line 5), and, for each dependent variable DV, an instrument is specified (Line 6). The argument is an attribute that uniquely identifies an object and is used in both execution and analysis to trace execution results to the objects that originated them. The instrument defines how to measure the corresponding dependent variable in the infrastructure.

For example, in Listing 6, we present the abstract syntax of an experiment specification (the concrete syntax is presented in Section 4) comprising a research hypothesis rh_1 (Line 2); an experimental design d, which applies a cartesian product and repeats each execution 8 times (Line 3); two treatments, *featureFamily* (Line 4) and *featureProduct* (Line 5); an experimental object *lift* (Line 6); and a dependent variable *analysisTime* (Line 7). The research hypothesis rh_1 (Line 2) compares the dependent variable

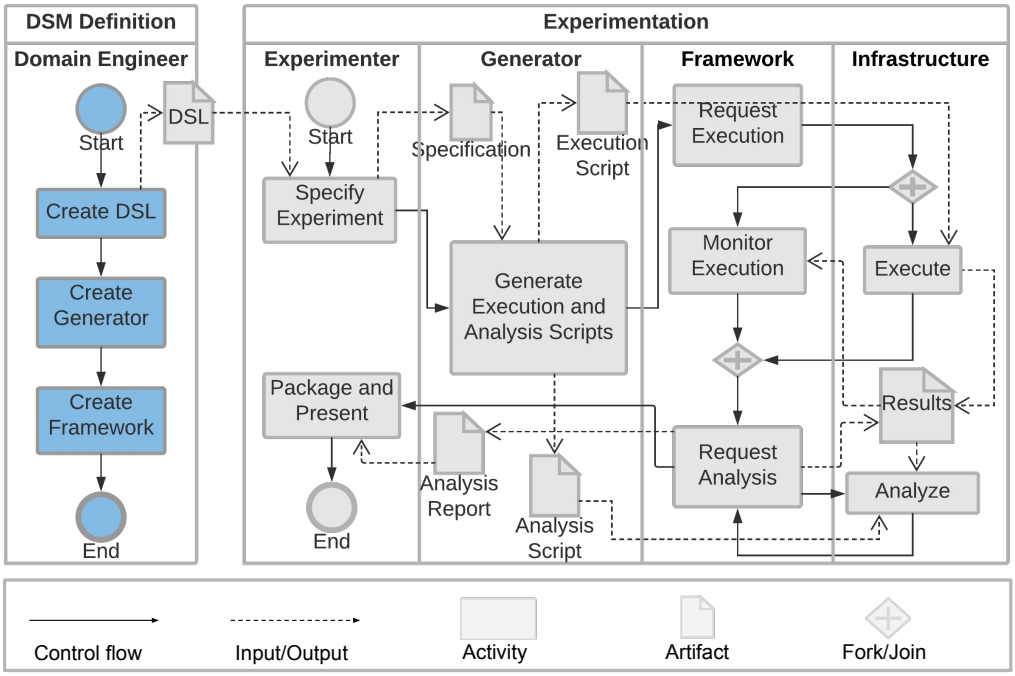


Fig. 1: Proposed DSM-based solution

Listing 5: DSL Syntax

1 $E ::= \{hypotheses : \overline{H},\ design : D,\ treatments : \overline{T},\ objects : \overline{O},\ dependentVariables : \overline{DV}\}$
2 $H ::= \{name : String,\ dependentVariable : DV,\ treatment_1 : T,\ treatment_2 : T\}$
3 $D ::= \{runs : PosInt,\ designFunction : \overline{T} \times \overline{O} \rightarrow \overline{(T, O)}\}$
4 $T ::= \{name : String,\ command : String\}$
5 $O ::= \{name : String,\ argument : String\}$
6 $DV ::= \{name : String,\ instrument : String\}$

analysisTime resulting from applying the treatments *featureFamily* and *featureProduct*.

Definition 1 (Experiment specification well-formedness) An experiment specification E is well-formed, denoted by $wf(e)$, if and only if all treatments and dependent variables referred to in its hypotheses are defined, and each hypothesis compares distinct treatments (Equation (1)). In addition, each hypothesis, treatment, object, and dependent variable is specified with a unique name; each treatment has a distinct valid command; each object has a distinct valid argument; and each dependent variable has a distinct valid instrument.

3.3 Experiment execution script generation and experiment execution

An execution script *ES* comprises a list of applications A (Line 1, Listing 7). Each application is defined by an instrument, related to a dependent variable, a command, related to a treatment, and an argument, related to an object (Line 2). In addition, an execution EX consists of a *dependentVariable*, a *treatment*, and an *object* (Line 3), whereas an execution result ER comprises the *instrument*, the *command*, and the *argument* used to run the application, and also the *value* resulting from its execution (Line 4).

Function *generateExecutionScript* (Line 1, Algorithm 1) uses as argument an experiment specification and generates the execution script ES. The first step is to apply function *applyDesign* (Line 5) using the experimental design, the hypotheses, and the objects defined in the experiment specification as arguments. It returns a duplicate-free list of executions EX. We use a duplicate-free list since a treatment can be referred to in more than one hypothesis. In this case, to prevent one treatment from being executed more times than another, there should be only one execution EX.

Listing 6: Example of an Experiment Specification

```
1  e = {[rh1], d, [featureFamily, featureProduct], [lift], [analysisTime]}
2  rh1 = {"rh1", analysisTime, featureFamily, featureProduct}
3  d = {8, cartesianProduct}
4  featureFamily = {"Feature Family", "FEATURE_FAMILY"}
5  featureProduct = {"Feature Product", "FEATURE_PRODUCT"}
6  lift = {"Lift", "lift"}
7  analysisTime = {"Analysis Time", "analysisTimeCommand"}
```

$$
\begin{aligned}
\forall e : E \cdot wf(e) \iff & (\forall h \in e.hypothesis \cdot \\
& \quad h.dependentVariable \in e.dependentVariables \land \\
& \quad h.treatment_1 \in e.treatments \land \\
& \quad h.treatment_2 \in e.treatments \land \\
& \quad h.treatment_1 \neq h.treatment_2) \\
& \land (\forall rh_1, rh_2 \in e.hypotheses \cdot rh_1 \neq rh_2 \implies \\
& \quad rh_1.name \neq rh_2.name) \\
& \land (\forall tr_1, tr_2 \in e.treatments \cdot tr_1 \neq tr_2 \implies \\
& \quad tr_1.name \neq tr_2.name \land tr_1.command \neq tr_2.command) \\
& \land (\forall o_1, o_2 \in e.objects \cdot o_1 \neq o_2 \implies \\
& \quad o_1.name \neq o_2.name \land o_1.argument \neq o_2.argument) \\
& \land (\forall dv_1, dv_2 \in e.dependentVariables \cdot dv_1 \neq dv_2 \implies \\
& \quad dv_1.name \neq dv_2.name \land dv_1.instrument \neq dv_2.instrument)
\end{aligned}
\tag{1}
$$

Listing 7: Execution Script and Execution Model

```
1  ES ::= {applications : A}
2  A ::= {instrument : String, command : String, argument : String}
3  EX ::= {dependentVariable : DV, treatment : T, object : O}
4  ER ::= {instrument : String, command : String, argument : String, value : Float}
```

From each execution (Lines 7–12), an application is generated by using *generateApplication* (Line 8). Then, each application is repeated the number of times defined in the experimental design (Lines 9–11). Finally, an execution script is created with all the generated applications (Lines 13–14).

Function *applyDesign* (Line 17) applies, for each hypothesis (Lines 19–30), *designFunction* to the treatments of that hypothesis and to the experimental objects (Line 22). This results in a series of treatment and object pairs related by the design function. From each pair (Lines 23–29), an execution *EX* is created (Line 24) using its treatment (Line 25), its object (Line 26), and the dependent variable of the corresponding hypothesis (Line 27).

Function *generateApplication* (Line 33) generates an application *A* from an execution *EX*. First, an application *A* is created (Line 34). Then, the command of the application is assigned with the corresponding command from the treatment of the execution (Line 35), the argument of the application is assigned with the corresponding argument from the object of the execution (Line 36), and the instrument of the application is assigned with the corresponding instrument from the dependent variable of the execution (Line 37).

For example, in Listing 8, we present the abstract syntax of an execution script generated from Listing 6, which contains applications (Lines 2 and 3) required to evaluate the research hypothesis *rh1* (Line 2, Listing 6). Each application is repeated 8 times, which is the number of runs defined in design (Line 3, Listing 6). For the sake of brevity, we omitted that repetitions.

Definition 2 (Execution script well-formedness) Every execution script is well-formed.

$$\forall es : ES \cdot wf(es)$$

The generation of the execution script must assure that, given a well-formed experiment specification (Definition 1), the resulting execution script is also well-formed (Definition 2):

Listing 8: Example of an Experiment Execution Script

```
es = [
        {"analysisTimeCommand", "FEATURE_FAMILY", "lift"},
        {"analysisTimeCommand", "FEATURE_PRODUCT", "lift"}
]
```

Algorithm 1 Execution Script Generation

```
1: function generateExecutionScript(experimentSpecification : E) : ES
2:     design ← experimentSpecification.design
3:     hypotheses ← experimentSpecification.hypotheses
4:     objects ← experimentSpecification.objects
5:     executions ← applyDesign(design, hypotheses, objects)
6:     applications ← new List
7:     for all execution ∈ executions do
8:         application ← generateApplication(execution)
9:         for i ← 1, design.runs do          ▷ Repeats execution design.runs times
10:            insert application into applications
11:        end for
12:    end for
13:    executionScript ← new ES
14:    executionScript.applications ← applications
15:    return executionScript
16: end function

17: function applyDesign(design : D, hypotheses : H̄, objects : Ō) : ĒX
18:    executions ← new List
19:    for all hypothesis ∈ hypotheses do
20:        t1 ← hypothesis.treatment1
21:        t2 ← hypothesis.treatment2
22:        relTreatmentsAndObjects ← design.designFunction({t1, t2}, objects)
23:        for all pairTreatmentObject ∈ relTreatmentsAndObjects do
24:            execution ← new EX
25:            execution.treatment ← pairTreatmentObject.treatment
26:            execution.object ← pairTreatmentObject.object
27:            execution.dependentVariable ← hypothesis.dependentVariable
28:            insert execution into executions
29:        end for
30:    end for
31:    return executions
32: end function

33: function generateApplication(execution : EX) : A
34:    application ← new A
35:    application.command ← execution.treatment.command
36:    application.argument ← execution.object.argument
37:    application.instrument ← execution.dependentVariable.instrument
38:    return application
39: end function
```

Property 1 (Execution script generation well-formedness)

The result of generating an execution script from a well-formed experiment specification is a well-formed execution script.

$$\forall e : E \cdot wf(e) \implies wf(generateExecutionScript(e))$$

Proof. By definition of *generateExecutionScript*, since every execution script *ES* is well-formed (Definition 2). □

To ensure soundness, in addition, Property 2 states that the generation of the execution script must assure that this script includes the applications required to evaluate all the research hypotheses defined in the experiment specification, and that each application is run the number of times defined in the experimental design.

Property 2 (Execution script generation soundness)

The infrastructure runs the required commands to execute a well-formed experiment (Equation (2)). Specif-

ically, for each hypothesis of a well-formed experiment, its treatments are applied n times to each experimental object, according to the experimental design and using the corresponding instrumentation. The number of repetitions n is specified in the experimental design.

Proof. By definition of *generateExecutionScript*, as it calls *applyDesign* and, for each hypothesis (Line 19, Algorithm 1), *applyDesign* applies the design of the experiment to the treatments related to the hypothesis and to the objects defined in the experiment (Line 22), resulting in pairs of related treatments and objects. When *generateExecutionScript* calls *generateApplication*, the resulting pairs of treatments and objects, together with the related dependent variable (Line 27), are mapped to their corresponding command (Line 35), argument (Line 36), and instrument (Line 37); the resulting application is repeated the number of times defined in the experimental design (Line 9). □

Furthermore, in addition to soundness, it is essential to optimize resource allocation, since experiment execution is often costly. In this vein, Property 3 states that the generated execution script contains only applications related to the hypotheses defined in the experiment specification.

Property 3 (Execution resource optimization)
The infrastructure runs only commands required to evaluate the hypotheses according to the design of the experiment, nothing else, as can be seen in Equation (3). Specifically, each application executed by the infrastructure maps to an execution of a treatment on an experimental object related to some dependent variable and hypothesis of the experiment. The treatment is related to one hypothesis specified in the experiment, and the instrument used to measure the dependent variable is related to the same hypothesis. In addition, the experimental object is related to the treatment according to the experimental design.

Proof. By definition of *generateExecutionScript*, as each application A is generated (Line 8, Algorithm 1) from an execution E resulting from applying the design of the experiment (Line 5) to the treatments of each hypothesis and to the experimental objects. □

After execution script generation, the supporting framework uses the function *execute* (Line 1, Algorithm 2) to request the running infrastructure to run the execution script and, then, collects a series of execution results *ER*. Each application in the execution script (Lines 3–11) is executed by the running infrastructure, and the return value is collected (Line 4). An execution result *ER* is created (Line 5), and the instrument (Line 6), the command (Line 7), and the argument (Line 8) used to run that application are assigned to the execution result; the value resulting from execution is assigned to field value (Line 9). Carrying over all four elements into the execution result is necessary for filtering purposes during analysis (Section 3.4).

The infrastructure semantics (Definition 3) consists of the results of executing the execution script in the running infrastructure.

Definition 3 (Infrastructure semantics)

$$\forall es : ES \cdot wf(es) \implies [\![es]\!] = execute(es)$$

3.4 Analysis script generation and analysis

An analysis script AS (Line 1, Listing 9) comprises a sequence of hypotheses tests $\overline{HT}$, each of which (Line 2) is defined by a *hypothesisName* and a sequence of analysis tests $\overline{AT}$. A hypothesis test is applied to each hypothesis, whereas an analysis test is applied to each object related by design function to the treatments of that hypothesis. Each analysis test AT (Line 3) is defined by an analysis function and two parameters P. These parameters (Line 4) are records with fields *instrument*, *command*, and *argument*. They are used to filter the execution results corresponding to the application that generated the result (cf. Algorithm 4, as explained later). Each hypothesis result HR (Line 5) is the result of analyzing each hypothesis and is defined by a *hypothesisName* and a sequence of *testResults*. Each test result TR (Line 6) is the result of the analysis test applied to the corresponding object. The *argument* is used to trace the test results to the corresponding object, and the *analysisResult* is the result of applying the analysis test. The analysis result AR (Line 7) contains a *String result* representing the result of the analysis test.

Function *generateAnalysisScript* (Line 1, Algorithm 3) generates the analysis script AS based on an experiment specification E. For each hypothesis (Lines 6–9) defined in the experiment specification, function *generateHypothesisTest* (Line 7) generates a hypothesis test using the experimental design, the corresponding hypothesis, and the experimental objects defined in the experiment specification. Finally, an analysis script is created (Line 10), and the generated hypotheses tests are assigned to field *hypothesesTests* (Line 11).

Function *generateHypothesisTest* (Line 14) generates a hypothesis test from the experimental design, a hypothesis, and a list of objects. It first calls *applyDesign* (Line 16), which results in a set of executions. Each execution comprises the dependent

$$\begin{aligned}\forall e : E \cdot wf(e) \implies & \forall h \in e.hypotheses \cdot \\ & \forall\ (t, o) \in e.design.designFunction(\{t_1, t_2\}, e.objects) \cdot \\ & \exists_{=n} a \in generateExecutionScript(e).applications \\ & \mid a.instrument = h.dependentVariable.instrument \\ & \wedge\ a.command = t.command \\ & \wedge\ a.argument = o.argument \end{aligned} \tag{2}$$

where

$t_1 = h.treatment_1$

$t_2 = h.treatment_2$

$$\begin{aligned}\forall e : E \cdot wf(e) \implies & \forall a \in generateExecutionScript(e).applications \cdot \\ & \exists h \in e.hypotheses, t \in \{t_1, t_2\}, o \in e.objects \\ & \mid a.instrument = h.dependentVariable.instrument \\ & \wedge\ a.command = t.command \\ & \wedge\ a.argument = o.argument \\ & \wedge\ (t, o) \in e.design.designFunction(\{t_1, t_2\}, e.objects) \end{aligned} \tag{3}$$

where

$t_1 = h.treatment_1$

$t_2 = h.treatment_2$

Algorithm 2 Experiment Execution

```
 1: function execute(executionScript : ES) : ER
 2:     results ← new List
 3:     for all application ∈ executionScript.applications do
 4:         value ← executeApplication(application)        ▷ Executes the application in the infrastructure
 5:         result ← new ER
 6:         result.instrument ← application.instrument
 7:         result.command ← application.command
 8:         result.argument ← application.argument
 9:         result.value ← value
10:         insert result into results
11:     end for
12:     return results
13: end function
```

variable defined for the hypothesis, either $treatment_1$ or $treatment_2$ related to the same hypothesis, and an object, related to the treatment by the design function. For each execution (Lines 17–25), function *generateAnalysisTest* (Line 21) generates an analysis test using the corresponding object and the hypothesis. Since the analysis test compares the execution results of both treatments, when applied to an object, there must be only one analysis test per object related to a given hypothesis. For this reason, before generating the analysis test, we first check if a test has already been generated for that object (Line 20).

Function *generateAnalysisTest* (Line 31) generates an analysis test from a hypothesis and a related object. First, the analysis test is created (Line 35). Then, the analysis function is retrieved by calling *suitableFunction* (Line 36), which is an oracle embedding the statistician's knowledge to provide a suitable analysis function for a given research hypothesis [11, 26]. Since this analysis function is provided uniquely based on the hypothesis, it is actually a procedure with parametric and nonparametric tests, as well tests to check the assumptions to the parametric tests. During analysis, when execution results are available, the analysis test first checks if all assumptions are satisfied, and, if so, the parametric test is applied. Otherwise, another (nonparametric) test is applied. Next, successive calls to function *generateParameter* generate $parameter_1$ (Line 37) and $parameter_2$ (Line 38) using the *dependentVariable*, *object*, and $treatment_1$ and $treatment_2$ of the *hypothesis*, respectively.

Finally, function *generateParameter* (Line 41) generates an parameter P from a *dependentVariable*, a *treatment*, and an *object*. The instrument of the de-

Listing 9: Analysis Script and Analysis Model

$AS ::= \{hypothesesTests : \overline{HT}\}$
$HT ::= \{hypothesisName : String,\ analysisTests : \overline{AT}\}$
$AT ::= \{analysisFunction : \overline{ER} \times \overline{ER} \rightarrow AR,\ parameter_1 : P,\ parameter_2 : P\}$
$P ::= \{instrument : String,\ command : String,\ argument : String\}$
$HR ::= \{hypothesisName : String,\ testResults : \overline{TR}\}$
$TR ::= \{argument : String,\ analysisResult : AR\}$
$AR ::= \{result : String\}$

pendent variable, the command of the treatment, and the argument of the object are assigned, respectively, to the instrument (Line 43), the command (Line 44), and the argument (Line 45) of the parameter P.

For example, in Listing 10, we present the abstract syntax of an analysis script generated from Listing 6, which contains the hypothesis test required to evaluate the research hypothesis *rh1* (Line 2, Listing 6). The analysis test (Line 2) comprises an analysis function and two parameters: p_1 (Line 4) and p_2 (Line 5), which are used to filter the execution results corresponding to the measurement of the dependent variable *analysisTime* resulting from the application of treatments *Feature Family* and *Feature Product* to the object *Lift*.

Definition 4 (Analysis script well-formedness) An analysis script is well-formed if and only if each distinct hypothesis test refers to a distinct hypothesis; each analysis test compares distinct treatments but the same object and dependent variable; and, for each hypothesis, each analysis test is related to a distinct object, as defined in Equation (4).

Similar to the generation of execution scripts, the generation of the analysis script must assure that, given a well-formed experiment specification (cf. Definition 1), the resulting analysis script is also well-formed (cf. Definition 4):

Property 4 (Analysis script generation well-formedness)

The result of generating an analysis script from a well-formed experiment specification is a well-formed analysis script.

$$\forall e : E \cdot wf(e) \implies wf(generateAnalysisScript(e))$$

Proof. By definition of *generateAnalysisScript*, since each hypothesis test is generated from a distinct hypothesis (Line 7, Algorithm 3) using a distinct *hypothesisName* (Line 27), each analysis test is generated from a distinct object (Line 21), and the parameters of the analysis test are generated from the same dependent variable and object but with a distinct treatment (Lines 37 and 38), since the experiment is well-formed. □

The supporting framework uses function *analyze* (Line 1, Algorithm 4) to request the running infrastructure to analyze the execution results using the previously generated analysis script and returning a series of hypothesis results $\overline{HR}$. The execution results are analyzed by each *hypothesisTest* of the analysis script (Lines 3–6) by calling the function *analyzeHypothesis* (Line 4). This function (Line 9) applies all the *analysisTests* (Lines 11–14) of that *hypothesisTest*. Each *analysisTest* is applied by the function *applyAnalysisTest* (Line 12), which returns a *testResult* *TR*. Then, a *hypothesisResult* is created (Line 15), the *hypothesisName* is assigned to field *hypothesisName* (Line 16), and the *testResults* are assigned to field *testResults* (Line 17).

Function *applyAnalysisTest* (Line 20) performs the analysis test and returns a *testResult*. It first filters the execution results (Lines 21–22) corresponding to each treatment using the parameters defined in the analysis test. Then, the analysis function is applied (Line 23) to the execution results, returning an analysis result. Finally, a *testResult* is created, and the argument (Line 25) and the analysis result are set to it.

Function *filterResults* (Line 29) filters execution results based on the *instrument*, the *command*, and the *argument* defined for the argument. Each subset of the execution results corresponds to the measurements of a dependent variable resulting from applying each treatment of a hypothesis to an experimental object.

The overall result of an experiment is a sequence of hypothesis results. Each hypothesis result represents the answer to a research hypothesis evaluated for each object, according to the experimental design.

Definition 5 (Experiment semantics) The semantics of a well-formed experiment consists of the confirmation/rejection of its hypotheses (Equation (5)).

Listing 10: Example of an Experiment Analysis Script

```
1  as = [
2          {"rh1", [ {analysisFunction, p1, p2} ] }
3  ]
4  p1 = {"analysisTimeCommand", "FEATURE_FAMILY", "lift"}
5  p2 = {"analysisTimeCommand", "FEATURE_PRODUCT", "lift"}
```

Algorithm 3 Analysis Script Generation

```
 1: function generateAnalysisScript(experimentSpecification : E) : AS
 2:     hypothesesTests ← new List
 3:     design ← experimentSpecification.design
 4:     hypotheses ← experimentSpecification.hypotheses
 5:     objects ← experimentSpecification.objects
 6:     for all hypothesis ∈ hypotheses do
 7:         hypothesisTests ← generateHypothesisTest(design, hypothesis, objects)
 8:         insert hypothesisTests into hypothesesTests
 9:     end for
10:     analysisScript ← new AS
11:     analysisScript.hypothesesTests ← hypothesesTests
12:     return analysisScript
13: end function

14: function generateHypothesisTest(design : D, hypothesis : H, objects : O) : HT
15:     analysisTests ← new List
16:     executions ← applyDesign(design, {hypothesis}, objects)
17:     for all execution ∈ execution do
18:         object ← execution.object
19:         visitedObjects ← new List
20:         if object ∉ visitedObjects then          ▷ Creates only one analysis test per object
21:             analysisTest ← generateAnalysisTest(hypothesis, object)
22:             insert analysisTest into analysisTests
23:             insert object into visitedObjects
24:         end if
25:     end for
26:     hypothesisTest ← new HT
27:     hypothesisTest.hypothesisName ← hypothesis.name
28:     hypothesisTest.analysisTests ← analysisTests
29:     return hypothesisTest
30: end function

31: function generateAnalysisTest(hypothesis : H, object : O) : AT
32:     dv ← hypothesis.dependentVariable
33:     t1 ← hypothesis.treatment1
34:     t2 ← hypothesis.treatment2
35:     analysisTest ← new AT
36:     analysisTest.analysisFunction ← suitableFunction(hypothesis)
37:     analysisTest.parameter1 ← generateParameter(dv, t1, object)
38:     analysisTest.parameter2 ← generateParameter(dv, t2, object)
39:     return analysisTest
40: end function

41: function generateParameter(dependentVariable : DV, treatment : T, object : O) : P
42:     parameter ← new P
43:     parameter.instrument ← dependentVariable.instrument
44:     parameter.command ← treatment.command
45:     parameter.argument ← object.argument
46:     return parameter
47: end function
```

$$
\begin{aligned}
\forall as : AS \cdot wf(as) \iff & (\forall ht_1, ht_2 \in as.hypothesesTests \cdot \\
& \quad ht_1 \neq ht_2 \implies ht_1.hypothesisName \neq ht_2.hypothesisName) \\
& \wedge (\forall ht \in as.hypothesesTests \cdot (\forall at \in ht \cdot \\
& \quad at.parameter_1.instrument = at.parameter_2.instrument \\
& \quad \wedge\ at.parameter_1.argument = at.parameter_2.argument \\
& \quad \wedge\ at.parameter_1.command \neq at.parameter_2.command) \\
& \wedge (\forall at_1, at_2 \in ht \cdot at_1 \neq at_2 \implies \\
& \quad at_1.parameter_1.argument \neq at_2.parameter_1.argument))
\end{aligned}
\tag{4}
$$

Algorithm 4 Analysis

```
 1: function analyze(executionResults : ER̄, analysisScript : AS) : HR̄
 2:     hypothesesResults ← new List
 3:     for all hypothesisTest ∈ analysisScript.hypothesesTests do
 4:         hypothesisResults ← analyzeHypothesis(executionResults, hypothesisTest)
 5:         insert hypothesisResults into hypothesesResults
 6:     end for
 7:     return hypothesesResults
 8: end function

 9: function analyzeHypothesis(executionResults : ER̄, hypothesisTest : HT) : HR
10:     testResults ← new List
11:     for all analysisTest ∈ hypothesisTest.analysisTests do
12:         testResult ← applyAnalysisTest(executionResults, analysisTest)
13:         insert testResult into testResults
14:     end for
15:     hypothesisResults ← new HR
16:     hypothesisResults.hypothesisName ← hypothesisTest.hypothesisName
17:     hypothesisResults.testResults ← testResults
18:     return hypothesisResults
19: end function

20: function applyAnalysisTest(executionResults : ER̄, analysisTest : AT) : TR
21:     results1 ← filterResults(executionResults, analysisTest.parameter1)
22:     results2 ← filterResults(executionResults, analysisTest.parameter2)
23:     analysisResult ← analysisTest.analysisFunction(results1, results2)
24:     testResult ← new TR
25:     testResult.argument ← analysisTest.parameter1.argument
26:     testResult.analysisResult ← analysisResult
27:     return testResult
28: end function

29: function filterResults(results : ER̄, parameter : P) : ER̄
30:     filteredResults ← new List
31:     for all result ∈ results do
32:         if result.instrument = parameter.instrument ∧ result.command = parameter.command ∧ result.argument =
    parameter.argument then
33:             insert result into filteredResults
34:         end if
35:     end for
36:     return filteredResults
37: end function
```

$$
\begin{aligned}
& \forall e : E \cdot wf(e) \implies [\![e]\!] = analyze(executionResults, analysisScript) \\
& \text{where} \\
& \quad executionResults = execute(executionScript) \\
& \quad executionScript = generateExecutionScript(e) \\
& \quad analysisScript = generateAnalysisScript(e)
\end{aligned}
\tag{5}
$$

Finally, the overall process, which includes execution script generation, execution, analysis script generation, and analysis, must assure that the experiment semantics (Definition 5) is consistent with the experiment specification, addressing Problem 3.

Property 5 (Experiment soundness)

The analysis is performed by using a suitable analysis function for each hypothesis and using correct parameters in the correct order (Equation (6)). In addition, execution data are produced by executing a sound execution script generated from the experiment specification.

$HR \leftrightarrow H$ is the bijection between hypotheses results HR and hypotheses H induced by the forward composition of the inverse of functions *analyzeHypothesis* and *generateHypothesisTest*. Given a hypothesis result $hr : HR$, $h : H$ is its corresponding hypothesis.

Likewise, $TR \leftrightarrow hObjects$ is the bijection between test results TR and the objects resulting from applying the design function to the treatments of a given hypothesis and experimentation objects. The bijection is induced by the forward composition of the inverse of the functions *applyAnalysisTest* and *generateAnalysisTest*. We also use a helper function $objects : \overline{(T, O)} \rightarrow \overline{O}$.

Proof. Let $e \in E, as \in AS, at \in AT$. By definition of *applyAnalysisTest*, since it applies $at.analysisFunction$ (Line 23, Algorithm 4) to two subsets of the execution results, filtered by *filterResults* using parameters $at.parameter_1$ (Line 21, Algorithm 4) and $at.parameter_2$ (Line 22, Algorithm 4); $at.analysisFunction$ is a suitable function to analyze the hypothesis (Line 36, Algorithm 3). The parameters used to filter each subset of the results are generated from the same dependent variable and object, but each one using a different treatment of the same hypothesis (Lines 37 and 38, Algorithm 3).

□

We note that the value of the proofs of Properties 1–5 is that they provide formal evidence of the correctness of the corresponding properties, which altogether address Problem 3 in Section 2. In other words, such proofs guarantee that analysis results actually *trace* to the stated experimental hypotheses, which could otherwise be overlooked in manual or even automated approaches as Remark 3 and Example 5 in Section 2 point out. Additionally, we note that the definitions and the algorithms in Section 3 provide a precise specification from which a solution, namely ExpRunA, to Problems 1 and 2 in Section 2 is implemented.

3.5 Running infrastructure

The main functions of the running infrastructure are to execute and to analyze the experiment. It receives commands from the supporting framework to run the execution script, reports the execution status, and sends execution results back to the supporting framework. Likewise, the running infrastructure receives commands to run the analysis script and sends analysis results back to the supporting framework.

The running infrastructure must be able to run applications specified in an execution script; check and report execution status; and collect execution results. For example, to run the execution script presented in Listing 8, the running infrastructure must be able to run the computational tools corresponding to each application (*FEATURE_FAMILY* and *FEATURE_PRODUCT*), report execution status, and present the results measured by the corresponding instrumentation (*analysisTimeCommand*). In addition, the running infrastructure must apply each analysis test of an analysis script (Listing 10) and present the corresponding results. We present the running infrastructure in detail in Section 4.

3.6 Supporting framework

The supporting framework integrates the DSM components and provides the interface between the generated code and the running infrastructure. It also monitors execution, collects results, and presents the analysis results to the experimenter.

The sequence diagram in Figure 2 shows how the supporting framework interacts with the other elements of our DSM solution. By calling function *generateExecutionScript* (Line 1, Algorithm 1), the supporting framework requests the generator to generate the execution script from the experiment specification; likewise, by calling *generateAnalysisScript* (Line 1, Algorithm 3), the supporting framework requests the generation of the analysis script. By calling function *execute* (Line 1, Algorithm 2), the framework requests the running infrastructure to execute the corresponding execution script. While execution is running, the framework monitors and gathers partial results from the running infrastructure. After finishing execution, by calling the function *analyze* (Line 1, Algorithm 4), the supporting framework requests the running infrastructure to analyze the execution results using the previously generated analysis script. Finally, the supporting framework collects the analysis results and present them to the experimenter.

$$
\begin{aligned}
&\forall e : E \cdot wf(e) \implies \forall hr \in [\![e]\!] \cdot \forall tr \in hr \cdot \\
&\quad tr = suitableFunction(h)(parameter_1\, data, parameter_2\, data) \\
&\text{where} \\
&\quad parameter_1\, data = filterResults(executionResults, parameter_1) \\
&\quad parameter_2\, data = filterResults(executionResults, parameter_2) \\
&\quad executionResults = execute(generateExecutionScript(e)) \\
&\quad parameter_1 = (h.dependentVariable.instrument, h.treatment_1.command, \\
&\quad o.argument) \\
&\quad parameter_2 = (h.dependentVariable.instrument, h.treatment_2.command, \\
&\quad o.argument) \\
&\quad h = (HR \leftrightarrow H)hr \\
&\quad hObjects = e.design.designFunction(\{h.treatment_1, h.treatment_2\}, \\
&\quad e.objects).objects \\
&\quad o = (TR \leftrightarrow hObjects)tr
\end{aligned}
\tag{6}
$$

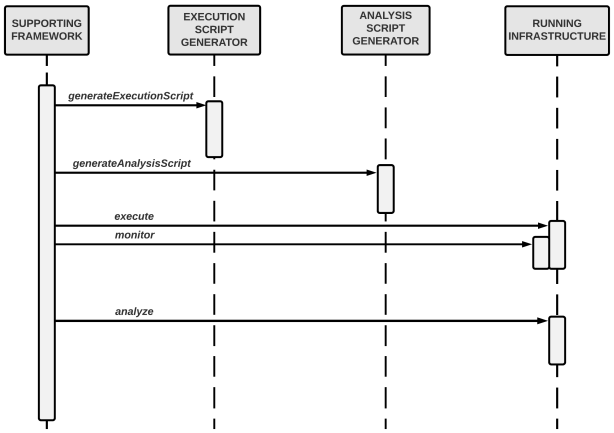


Fig. 2: Supporting framework interactions

4 *ExpRunA*

In this section, we present *ExpRunA*,[7] a Web-based tool that implements the DSM approach (Section 3), providing a means to specify executable specifications at a high level of abstraction, automated execution, data analysis, and results presentation. We present its functional view (Section 4.1), its architecture (Section 4.2), and its implementation (Section 4.3).

4.1 Functional view

To conduct an experiment using *ExpRunA*, an experimenter first must create an experiment specification using *ToExpDSL*. To ease this task, we created a specific editor with syntax highlighting, content assist, syntax validation, static semantics validation, template proposals, and text hover (Figure 3). When an experiment

[7] https://expruna.github.io/.

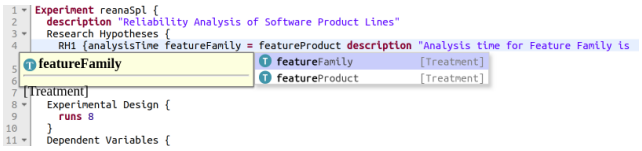


Fig. 3: DSL editor

is specified using the editor, its specification is type checked according to the grammar rules and static semantics validation rules. Non-conformity is reported as an error or as a warning by the editor.

Listing 11 shows a specification using our DSL, which was adapted from the original experiment conducted by Lanna et al. [30]. In this specification, the research hypothesis RH1 (Line 4) compares the dependent variable analysisTime for the treatments featureFamily and featureProduct. The dependent variable analysisTime (Line 10) has a corresponding instrumentation (Line 13). The instrumentation consists of a command and a value expression. The command is used to run the instrumentation tool, whereas the value expression is used to build a regular expression and to extract the corresponding value from the output. The treatments (Lines 18–21) are related to the factor strategy (Line 16). Each treatment defines a parameter argument and uses the execution reanaEvaluator (Lines 19 and 20). Each object defines a parameter spl (Lines 24 and 27). The execution reanaEvaluator (Lines 32–34) defines a command using the placeholders ${treatment.parameter.argument} and ${object.parameter.spl} (Line 33), which are replaced by the corresponding values defined for each treatment and object during the execution script generation.

After specifying the experiment, the experimenter can generate execution and analysis scripts by running the command Generate. The command Generate and Run (Figure 4) generates and runs the scripts. The exe-

Listing 11: Example of an experiment specification

```
1  Experiment reanaSpl {
2    description "Reliability Analysis of Software Product Lines"
3    Research Hypotheses {
4      RH1 {analysisTime featureFamily = featureProduct description "Analysis time for Feature Family is equal to analysis time for
           Feature Product"}
5    }
6    Experimental Design {
7      runs 8
8    }
9    Dependent Variables {
10     analysisTime { description "Analysis time" scaleType Absolute unit "ms" instrument analysisTimeCommand }
11   }
12   Instruments{
13     analysisTimeCommand {command "/usr/bin/time -v" valueExpression "Total analysis time:" }
14   }
15   Factors {
16     strategy { description "Analysis Strategy" scaleType Nominal}
17   }
18   Treatments {
19     featureFamily description "Feature Family" factor strategy parameters{argument "FEATURE_FAMILY"} execution
           reanaEvaluator,
20     featureProduct description "Feature Product" factor strategy parameters{argument "FEATURE_PRODUCT"} execution
           reanaEvaluator
21   }
22   Objects { description "SPL" scaleType Nominal {
23         lift {
24           description "Lift" parameters {spl "lift"}
25         },
26         intercloud {
27           description "Intercloud" parameters {spl "intercloud"}
28         }
29     }
30   }
31   Executions {
32     reanaEvaluator {
33       command "java -Xss100m -Xmx8g -jar reana-spl.jar --all-configurations --suppress-report --stats --param-path
           = param --analysis-strategy = ${treatment.parameter.argument} --feature-model =
           ${object.parameter.spl}/models/0.txt --uml-models = ${object.parameter.spl}/models/0_model.xml"
34     }
35   }
36 }
```

cution script is executed by the running infrastructure, and, during execution, the execution status is presented to the experimenter (Figure 5). Execution results are collected, and then analyzed by the analysis script. Finally, the experimenter can access not only a report containing plots, statistical tests, and the overall results of the experiment, but also the raw data and the generated scripts. The experimenter can also re-run the analysis using the command Run Analysis or perform additional analysis using the raw data and the scripts. The boxplot presented in Figure 6 corresponds to the analysis of RH1, which compares the analysis time of the treatments featureFamily and featureProduct (Line 4, Listing 11), for the experiment object Lift (Lines 23–24, Listing 11).

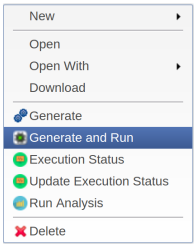


Fig. 4: Generate and run command

4.2 Architecture

ExpRunA's architecture is modular and extensible due to Eclipse's[8] extension mechanism. The core compo-

[8] https://www.eclipse.org/.

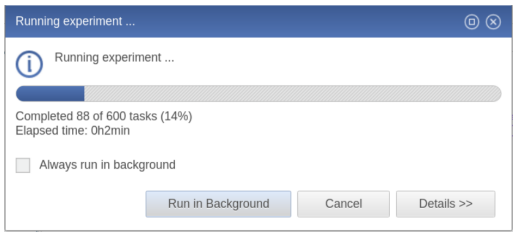


Fig. 5: Execution status

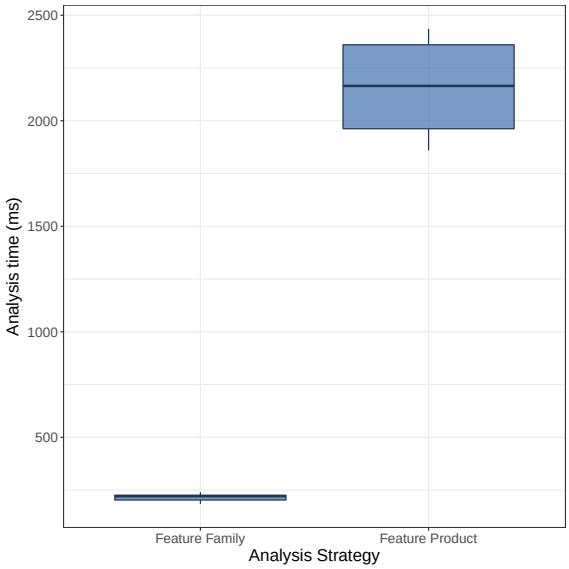


Fig. 6: Excerpt of an analysis report

nent comprises the grammar, the validators, and interfaces to define generators, commands, and access to a database (Figure 7). The execution script generator (DohkoGenerator) and the analysis script generator (RScriptGenerator) are implementations of IGenerator. Additional generators can be defined by implementing this interface. The commands that can be run from the supporting framework are defined by implementing interface ICommand. The component RunDohko implements the command to run the execution script, and the RunAnalysis implements the command to run the analysis script. The ExecutionStatus component interacts with the running infrastructure to monitor the execution status. The component MongoDBApi implements the access to database.

The running infrastructure must be able to run the execution script and the analysis script. In our exploratory studies, we identified Dohko [31] as a candidate solution to be used in *ExpRunA*, because it not only fulfills all the requirements presented in Section 3.5 but also provides self-configuration, self-healing, and scalability in inter-cloud environments. This frees the researcher from the often error-prone and time-consuming task of

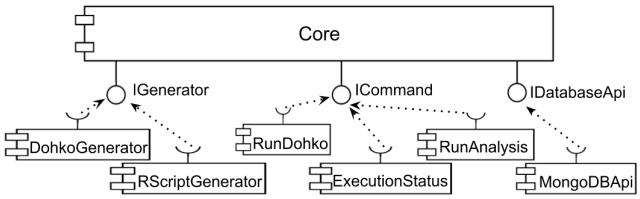


Fig. 7: Tool components

manually performing the configuration and initialization of the computing infrastructure with sufficient resources to run the experiments in a timely manner. Furthermore, Slurm [54] is a flexible and fault-tolerant cluster resource management system. It provides a simple, robust, and scalable parallel job execution environment for clusters. Both Dohko and Slurm could be used as infrastructure solution in our approach. We decided for Dohko since it can manage resources not only on clusters but also in inter-cloud environments. To run the analysis, we created an environment with R[9] for data analysis and LaTeX[10] for presentation of results. We actually run R Sweave scripts, which embed R code chunks in LaTeX documents. This is in line with the ideas of a Reproducible Research, as proposed by Madeyski and Kitchenham [33].

Each main component, that is, the supporting framework, execution environment, analysis environment, and database, is run in its own Docker container, which enables customization, distributed execution, environment isolation, and portability. Given the modularity of *ExpRunA*, each component can be easily replaced by a custom image. For example, the users can easily adapt their current infrastructure by creating a new Docker image based on the Dohko image we provide and installing their specific packages and dependencies on it. In highly resource-consuming experiments, distributed execution enables leveraging resources from multiple machines, achieving a greater performance than using a single machine. In addition, environment isolation prevents the other components from affecting execution results, specially when it comes to performance measurements, such as runtime and memory consumption. Finally, portability enables *ExpRunA* to be run in distinct environments, consequently, easing execution and replication of experiments.

[9] https://www.r-project.org.
[10] https://www.latex-project.org.

4.3 Implementation

Using Xtext,[11] we created *ToExpDSL* partially based on ExpDSL by Freire et al. [20]. ExpDSL comprises four views: process view, metric view, experimental plan view, and questionnaire view. The metric view and experimental view are the same for human-oriented and technology-oriented experiments; thus, they can be reused in our setting. Nevertheless, since the process view and the questionnaire view are bound to human-oriented experiments, they cannot be reused in our setting. So, we created *ToExpDSL* with new constructs for technology-oriented experiments, which supports the specification of execution parameters related to the treatments, as well infrastructure requirements, such as the number of CPUs and memory size. In addition, since both the grammar and the generated artifacts are significantly distinct from ExpDSL, we also developed our own code generators and supporting framework.

The concrete syntax of the grammar was specified in Xtext, which is a tool set for the definition of textual languages. In fact, there are distinct tools that support the definition of DSLs, either textual or graphical. For instance, Web Generic Modeling Environment (WebGME)[34] is a Web-based infrastructure to support graphical Domain-Specific Modeling Languages (DSML). However, we chose Xtext due to its integration with Eclipse and to our familiarity with it. *ToExpDSL*'s full grammar is presented on our supplementary Website. The parser of the *ToExpDSL* receives the specification of an experiment and returns a corresponding object. Then, the validators and code generators access this object and all its elements to, respectively, validate and generate the code.

The validators and code generators have been implemented in Xtend.[12] The validators complement those provided by the grammar rules to check the well-formedness (Definition 1) of the experiment specification. Additionally, the validators report warnings whenever a dependent variable, a factor, a treatment, or an execution is never used. Our validation rules are listed on our supplementary Website.

After validating the specification, the code generators access the experiment model and, leveraging string templates, generate code. We implemented two code generators: an executions script generator and an analysis script generator.

Since we are using Dohko as infrastructure, the execution script is actually a Dohko Application Descriptor. Listing 12 shows an excerpt of the generated execution script corresponding to the experiment specification in Listing 11. According to Algorithm 1, the execution script generator applies the treatments to the objects according to the design function, which expresses how treatments relate to objects. Currently, *ExpRunA* supports only design functions expressed as any subset of a Cartesian product of treatments and objects. For instance, the experimenter could restrict the application of the treatment featureProduct only to the object Lift. Since no restriction was applied to the design in our example (Lines 6–8, Listing 11), a Cartesian product is used to relate the treatments to the objects. Accordingly, each block of applications in the execution script corresponds to the application of a treatment to an object (Lines 5–7, 8–10, 11–13, and 14–16, Listing 12). The command line of each application is generated by combining the instrumentation command and the execution command. Furthermore, the placeholders related to treatments and objects are replaced by the corresponding values. For instance, by applying the treatment featureFamily to the object lift, the resulting command Line (Line 7) uses the instrumentation command (Line 13, Listing 11) related to the dependent variable analysisTime, the command line defined for reanaEvaluator (Line 33, Listing 11), the parameter argument defined for the treatment featureFamily (Line 19, Listing 11), and the parameter spl defined for the object lift (Line 24, Listing 11). Finally, the resulting application is repeated the number of times defined by runs (Line 7, Listing 11). For the sake of brevity, we omitted these repetitions in Listing 12.

Listing 12: Excerpt of a generated execution script corresponding to the experiment specification in Listing 11

```
1  ---
2  name: "reanaSpl"
3  description: "Reliability Analysis of Software Product Lines"
4  blocks:
5    - applications:
6      - name: "featureFamily_lift_0"
7        command-line: "/usr/bin/time -v java -Xss100m
                 -Xmx8g -jar reana-spl.jar
                 --all-configurations --suppress-report
                 --stats --param-path = param
                 --analysis-strategy = FEATURE_FAMILY
                 --feature-model = lift/models/0.txt
                 --uml-models = lift/models/0_model.xml"
8    - applications:
9      - name: "featureFamily_intercloud_0"
10       command-line: "/usr/bin/time -v java -Xss100m
                 -Xmx8g -jar reana-spl.jar
                 --all-configurations --suppress-report
                 --stats --param-path = param
                 --analysis-strategy = FEATURE_FAMILY
                 --feature-model = intercloud/models/0.txt
                 --uml-models =
                 intercloud/models/0_model.xml"
11   - applications:
```

[11] https://eclipse.org/Xtext/.
[12] http://www.eclipse.org/xtend/.

```
12      - name: "featureProduct_lift_0"
13        command-line: "/usr/bin/time -v java -Xss100m
                -Xmx8g -jar reana-spl.jar
                --all-configurations --suppress-report
                --stats --param-path = param
                --analysis-strategy = FEATURE_PRODUCT
                --feature-model = lift/models/0.txt
                --uml-models = lift/models/0_model.xml"
14    - applications:
15      - name: "featureProduct_intercloud_0"
16        command-line: "/usr/bin/time -v java -Xss100m
                -Xmx8g -jar reana-spl.jar
                --all-configurations --suppress-report
                --stats --param-path = param
                --analysis-strategy = FEATURE_PRODUCT
                --feature-model = intercloud/models/0.txt
                --uml-models =
                intercloud/models/0_model.xml"
```

The corresponding generated analysis script is an R Sweave script (Listing 13). The analysis script starts with ordinary LaTeX code (Lines 1–5). For each hypothesis (Line 4), and for each object related to the treatments of that hypothesis by the design function (Line 5), the analysis is performed with R code (Lines 6–24). First, a boxplot is generated (Lines 7–15) from the execution results corresponding to the dependent variable and treatments related to the hypothesis, and the experimental object at hand (Lines 7 and 9). Using the same subset of the execution results, the analysis test first checks whether the assumptions to apply a parametric test are satisfied (Line 17). If so, it applies a parametric test (Line 18) and presents the results (Line 19). Otherwise, it applies a nonparametric test (Line 21) and then presents the results (Line 22).

Listing 13: Excerpt of a generated analysis script

```
1  \begin{document}
2  \title{Reliability Analysis of Software Product Lines}
3  \section{Research Hypotheses}
4  \subsection{RH1: Analysis time for Feature Family is equal to
       analysis time for Feature Product}
5  \subsubsection{RH1.1: Object Lift}
6   <<RH1_lift, include=TRUE, echo=FALSE,
       warning=FALSE, message=FALSE >> =
7   DF = subset(json_data, (treatment == 'featureFamily' |
       treatment == 'featureProduct') & object == 'lift')
8   DF$treatmentDescription =
       ordered(DF$treatmentDescription, levels =
       levels(DF$treatmentDescription)[
       order(as.numeric(by(DF$analysisTime,
       DF$treatmentDescription, mean)))])
9   boxplot_RH1_lift = ggplot(DF, aes(x
       =treatmentDescription , y = analysisTime)) +
10    geom_boxplot(fill = "#4271AE", colour =
          "#1F3552",alpha = 0.7,outlier.colour =
          "#1F3552", outlier.shape = 20)+
11    theme_bw() +
12    scale_x_discrete(name = "Analysis Strategy")+
13    ggtitle("Analysis time by Analysis Strategy for Lift") +
14    ylab("Analysis time (ms)")
15  boxplot_RH1_lift
16
17  if(shap_featureFamily_lift$p.value > alpha &
        shap_featureProduct_lift$p.value > alpha){
18    tTest = t.test(subset(json_data, treatment ==
          'featureFamily' & object == 'lift')$analysisTime,
          subset(json_data, treatment == 'featureProduct' &
          object == 'lift')$analysisTime, var.equal =
          fTest$p.value > alpha, paired = FALSE)
19    print(tTest)
20  }else{
21    wTest = wilcox.test(analysisTime~treatment,
          data=subset(json_data, (treatment ==
          'featureFamily' | treatment == 'featureProduct') &
          object == 'lift'))
22    print(wTest)
23  }
24  @
25  \end{document}
```

If it is necessary to change the definition of any element of *ToExpDSL*, or to add elements to the grammar (e.g., to specify additional design types), it is possible to change the grammar defined in file AutoExp.xtext. It is also possible to create a new grammar definition that extends *ToExpDSL*. Xtext's extension mechanism allows us to add or override grammar definitions. It is important to note, for changes in the grammar to be effective, the code generators, and occasionally the validators, must be changed accordingly. For example, suppose we want to compare k treatments in a fully randomized design [11]. We would have to change the ResearchHypothesisFormula element in AutoExp.xtext to support the definition of more than two treatments. Then, we would have to change the DohkoGenerator class to include all the applications in the execution script corresponding to the treatments defined for each hypothesis. Likewise, we would have to change the RScriptGenerator class to include suitable analysis tests to compare k treatments in a fully randomized design, such as Analysis of Variance (ANOVA) [11]. Finally, we also would have to change the validator to check, for instance, if there are repeated treatments in the hypothesis definition.

Additional rules in the validators can be created by adding methods to the class AutoExpValidator and annotating them with @Check. The validation method must define the element of the DSL that will be checked as a parameter. Inside the validation method, it is possible to verify the element and throw a warning or an error, in case there is a non-compliance.

Using DSLFORGE [29], an initial version of *ExpRunA* was automatically generated from *ToExpDSL*'s grammar and code generators. The generated application is based on Eclipse Remote Application Platform (RAP) and includes the Web editor and commands to create and delete models, and also to generate code from the model. We extended and customized this initial ver-

sion of *ExpRunA* with additional commands to enable execution, monitoring, data analysis, and presentation of results.

5 Evaluation

We formally proved that our model complies with key correctness properties to assure that execution and analysis results are correct according to the experiment specification (Sections 3.3 and 3.4). In this section, we empirically assess our solution with respect to automation, level of abstraction, and correctness. Correspondingly, in Section 5.1, we investigate the expressiveness of our tool to specify technology-oriented experiments (RQ 1) and whether it can automate execution and analysis from the specification of technology-oriented experiments (RQ 2 and RQ 3). We then evaluate the level of abstraction (Section 5.2) by comparing specifications of previously published experiments and specifications using our DSL (RQ 4), and by comparing DSL's grammar constructs with experimentation concepts (RQ 5). Finally, in Section 5.3, we analyze and interpret the assessment results, discussing threats to validity, limitations of the contribution, and future work.

5.1 Execution and analysis automation

The main goal of this section is to assess the ability of *ExpRunA* to provide automation in the experimentation process and is guided by the following research questions:

RQ 1. Can *ExpRunA* be used to specify technology-oriented experiments?

RQ 2. Does *ExpRunA* facilitate sound automation of experiment execution from the specification of technology-oriented experiments?

RQ 3. Does *ExpRunA* facilitate sound automation of experiment analysis from the specification of technology-oriented experiments?

5.1.1 Evaluation method

To address RQs 1 to 3, we first randomly selected three previously published experiments[13] meeting the criteria described in Section 5.1.2. For each experiment, we performed two replications: one using *ExpRunA* and another using the scripts provided by the original authors. With *ExpRunA*, we specified each experiment using *ToExpDSL*, which assesses whether *ToExpDSL* is sufficiently expressive to specify technology-oriented experiments (RQ 1). Since the main goal of the evaluation is to assess the feasibility of *ExpRunA*, not usability, we used *ToExpDSL* ourselves (as future work, we plan an independent usability evaluation). Then, we used *ExpRunA* starting from specification, to generate and execute execution and analysis scripts. We also replicated the experiments using the original scripts and, then, compared the results with the results obtained with *ExpRunA*. This way, we assure that not only *ExpRunA* can generate execution and analysis scripts, but also that these scripts can produce sound results. With *sound results*, we mean execution results that lead to the same conclusions as the original results.

To evaluate *ExpRunA*, we conducted external replications, with no interaction with original experimenters. We used the published papers and the laboratory packages provided by the authors. The replications were as similar as possible to the original experiments, except for the machines. For practical reasons, we used the same machine type for all experiments, without taking into account the original machine resources. This may affect the absolute execution time but should not affect the overall conclusions of the experiments. In some cases, we also made some minor changes in the original scripts to ease execution and data collection. For instance, we saved execution results in a file instead of showing them in the console. These changes did not affect how the experiment is executed and analyzed, though.

All the experiments were run on Google Cloud Platform on a virtual machine type *n1-standard-4* running Ubuntu 16.10. The machine has 4 CPUs and 15 GB RAM. To keep the execution environment as similar as possible, both replications were run inside the same Docker container and running in the same virtual machine. The complete specification, scripts files and results, as well instructions for future replications can be obtained from our supplementary Website.

5.1.2 Experiments selection

We selected three experiments meeting the following criteria:

– The experiment is reported in a paper published in a venue explicitly requiring reproducibility as part of the evaluation process or highlighting this fact in accepted papers. The venues considered were International Conference on Computer Aided Verification (CAV) and Joint Meeting on Foundations of Software Engineering (FSE).

[13] The set of experiments we found is presented on our supplementary Website.

– The experiment is technology-oriented, that is, a software, instead of a person, applies treatments to objects.
– Replication is completely documented.
– Every software, script, and artifact required to replicate the experiment is publicly available.
– Each hypothesis of the experiment compares two treatments of the same factor at a time, or the experiment can be decomposed in pairwise comparisons.

Based on these criteria, we selected the following experiments:

Experiment 1. Bak and Duggirala [4] presented a technique to perform simulation-equivalent reachability and safety verification of linear systems with inputs. To evaluate their proposal, they created a tool named Hylaa (HYbrid Linear Automata Analyzer).[14] In their evaluation, the authors examined the effects of optimizations for computing reachability for linear time-invariant systems with inputs. They compared a basic algorithm (Basic), warm-start optimization (Warm), Minkowski sum decomposition (Decomp), and Hylaa (uses both Minkowski sum decomposition and warm-start). Measurements for the no-input system (NoInput) were included for reference and could be considered a lower bound for the simulation-based methods if the time to handle the inputs could be completely eliminated. In order to measure runtime, the number of steps in the problem was varied by changing the step size and keeping the time bound fixed at 2π. Then, runtime for each optimization was measured, recording 10 measurements in each case. The performance of the basic algorithm (Basic) is improved by warm-start optimization (Warm), but not as much as when the Minkowski sum decomposition optimization is used (Decomp). Combining both optimizations works even better (Hylaa). The reachability time for the system without inputs (NoInput) makes a lower bound.

Experiment 2. Brennan et al. [12] presented a constraint caching framework to expedite potentially expensive satisfiability and model-counting queries. Their techniques were implemented in a tool named Cashew,[15] which was built as an extension of the Green caching framework [49]. Cashew was also integrated with Symbolic PathFinder (SPF) [37] and the ABC [3] model-counting constraint solver. The authors investigated the effects of their normalization procedure on model-counting datasets of string constraints. The Kaluza dataset [42], a well-known benchmark of string constraints, was used in their evaluation. This dataset contains 1,342 big constraints (SMC-Big) and 17,554 small constraints (SMC-Small). Another version of this dataset (without duplicates), with 359 constraints in SMC-Big and 9,745 constraints in SMC-Small, was also used. The results of model-counting all constraints in each set (SMC-Big and SMC-Small, original and without duplicates) are presented in Table 1. Table 1 shows that, on the SMC-Big set without duplicates, Cashew achieved a speedup of over 10x, and, on the SMC-Small set without duplicates, of 2.19x. For the original datasets, the speedup was 89.70x on SMC-Big, and 2.60x on SMC-Small. The authors remarked that the speedup on SMC-Big original dataset is due to the presence of duplicates, which makes even caching with no normalization very effective. They also investigated the effect of disabling each transformation in the normalization procedure. Table 2 shows the number of orbits that are achieved by different subsets of the transformations. The number of orbits represents the number of distinct constraints resulting from applying the normalization procedure. The removeVar and removeConj transformations are preprocessing steps that remove redundant variables and conjuncts, respectively. The other transformations are re-ordering (σ_I), renaming the variables ($\sigma_{\mathbb{V}}$), and permuting the alphabet constants (σ_{Σ}). The results indicate that all transformations yield some benefit, and that $\sigma_{\mathbb{V}}$ is the most beneficial transformation.

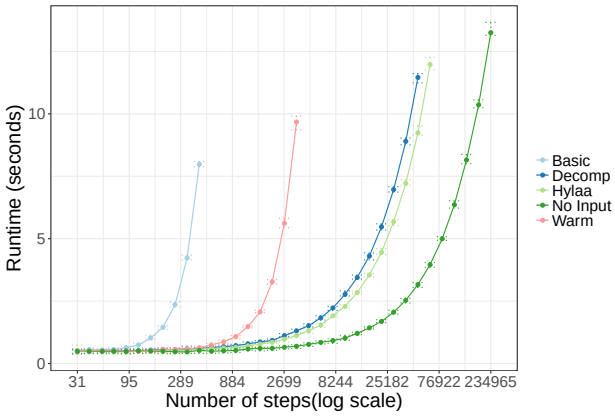
Experiment 3. The third experiment is the second part of the experimental evaluation presented in Brennan et al. [12]. In this experiment, the authors investigated the effects of their normalization procedure on side-channel analysis. They used Symbolic PathFinder [37] with Cashew to symbolically execute four Java programs that operate on strings: Password1, Password2, Obscure, and CRIME. Password1 contains a method that checks whether a user-given string matches a secret password. Password2 is a variant of the previous one that requires a certain number of characters to be compared before returning, even if a mismatch has already been found. Obscure is a Java translation of the *obscure.c* program used in Luu et al. [32], which is a password change authorizer. CRIME is a Java version of a well-known attack, Compression Ratio Info-leak Made Easy [5, 40]. For each of the four programs, they ran 1,000 symbolic-execution-based side-channel analyses, using as the secret each of the 1,000 passwords in the *RockYou1K* dataset [52]. Table 5 shows execution time, hits and misses for three execution modes. The first mode uses neither normalization nor caching. In the second mode, only caching without normalization is performed. In the third mode, Cashew's normalization is enabled. The results show that Cashew achieved an average speedup of nearly 3x, while caching without

[14] https://bit.ly/2NTCuSe.
[15] https://github.com/vlab-cs-ucsb/cashew/.

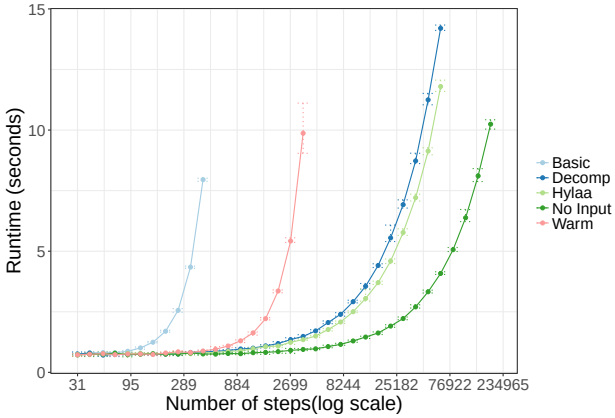
normalization achieved only 1.06x. The hit/miss ratios improve dramatically when switching to Cashew.

5.1.3 Results

We present the results of replicating Experiments 1 to 3. The results of replicating the Experiment 1 with *ExpRunA* (Figure 8b) are consistent with the replication using original scripts (Figure 8a) and with the results presented in the original paper (Figure 3 [4]): Basic is the worst performing variant, followed by Warm and Decomp; Hylaa is better than Decomp; and NoInput is a lower bound. The relative differences between the results with *ExpRunA* and with original scripts are presented in Figure 9. The differences are comparatively high for runtime values below 1 s, reaching more than 70% for NoInput. However, the differences decrease quickly to nearly 20% for 1 s, to 10% for two seconds, and to 5% for 3 s. Above 3 s, the differences keep below 5%.



(a) Original scripts



(b) *ExpRunA*

Fig. 8: Results of replicating Experiment 1

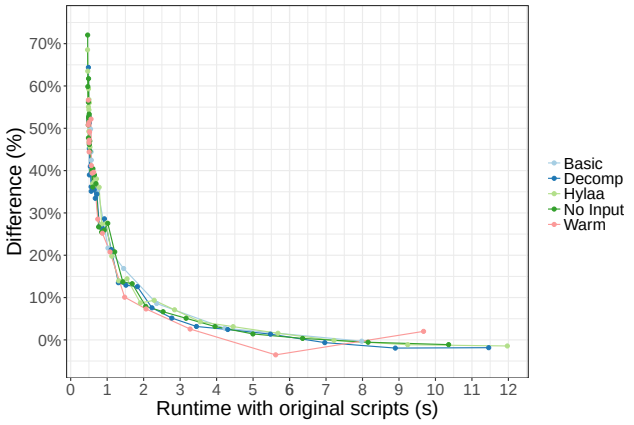


Fig. 9: Relative differences between replications with *ExpRunA* and with original scripts for Experiment 1

The second replicated experiment is Experiment 2. Due to the high number of duplicates present in original dataset and to avoid an excessive time-consuming experiment, we used only the Kaluza dataset without duplicate in our replications. The results are presented in Tables 3 and 4. Using the original scripts, Cashew achieved a speedup of 20.62x on the SMC-Big, and 2.43x on SMC-Small; using *ExpRunA*, 26.06x on SMC-Big and 24.45x on SMC-Small. When it comes to the effect of each transformation in the normalization procedure, the results of each replication are exactly the same. These results are consistent with the results presented in the paper (Tables 1 and 2). The only difference is the number of orbits on SMC-Small with no transformation, which is 9710 for both replications, whereas the number presented in the paper is 9754. Originally, the authors computed the hash of each constraint file and removed duplicates. They assumed that the number of unique constraint files would be the same as the number of orbits when no transformations were enabled; however, this assumption was incorrect due to the different variable declarations in files with the exact same constraint. The relative differences between the average runtime results with *ExpRunA* and with original scripts are presented in Figure 10. The difference is around 30% for SMC-Big without caching and below 5% for the other cases.

The last replicated experiment was Experiment 3. The results are presented in Tables 6 and 7. Cashew achieved an average speedup of 2.8x (original) and 2.43x (*ExpRunA*), while caching without normalization achieved 1.07x (original) and 1.08x (*ExpRunA*). The number of hits and misses for all programs is exactly the same for both replications. These results are consistent with the results presented in the original paper (Table 5). However, there is a difference in results

Table 1: Results of Model counting SMC-Big and SMC-Small [12]

		Wihtout caching	With caching	Speedup
Big (no dups)	Average	8.94 s	0.82 s	10.90x
	Maximum	121.92 s	40.13 s	3.03x
	Total time	3,208.65 s	293.21 s	10.94x
Small (no dups)	Average	0.12 s	0.05 s	2.40x
	Maximum	1.09 s	1.12 s	0.97x
	Total time	1,211.09 s	552.56 s	2.19x
Big (original)	Average	23.32 s	0.26 s	89.70x
	Maximum	121.92 s	40.13 s	3.03x
	Total time	31,297.90 s	358.17 s	87.38x
Small (original)	Average	0.13 s	0.05 s	2.60x
	Maximum	1.09 s	1.12 s	0.97x
	Total time	2,221.91 s	971.50 s	2.29x

Table 2: Effect of transformations on orbit refinement [12]

Transformations enabled	#Orbits (SMC-Big)	#Orbits (SMC-Small)
None	359	9754
All transformations	34	360
All except σ_I	72	376
All except $\sigma_{\mathbb{V}}$	344	9645
All except σ_{Σ}	35	841
All except removeVar	34	361
All except removeConj	40	386

Table 3: Results of Model counting SMC-Big and SMC-Small (replication)

		Original scripts			*ExpRunA*		
		Without caching	With caching	Speedup	Without caching	With caching	Speedup
Big	Avg	12.94 s	0.63 s	20.62x	16.79 s	0.64 s	26.06x
	Max	178.64 s	17.35 s	10.30x	273.96 s	17.55 s	15.61x
	Total	4,645.99 s	217.78 s	21.33x	6,028.07 s	223.57 s	26.96x
Small	Avg	0.21 s	0.09 s	2.43x	0.22 s	0.09 s	2.45x
	Max	1.42 s	1.57 s	0.90x	1.45 s	1.55 s	0.93x
	Total	2,070.63 s	853.79 s	2.43x	2168.72 s	885.42 s	2.45x

Table 4: Effect of transformations on orbit refinement (replication)

	Original scripts		*ExpRunA*	
Transformations enabled	#Orbits (SMC-Big)	#Orbits (SMC-Small)	#Orbits (SMC-Big)	#Orbits (SMC-Small)
None	359	9710	359	9710
All transformations	34	360	34	360
All except σ_I	72	376	72	376
All except $\sigma_{\mathbb{V}}$	344	9645	344	9645
All except σ_{Σ}	35	841	35	841
All except removeVar	34	361	34	361
All except removeConj	40	386	40	386

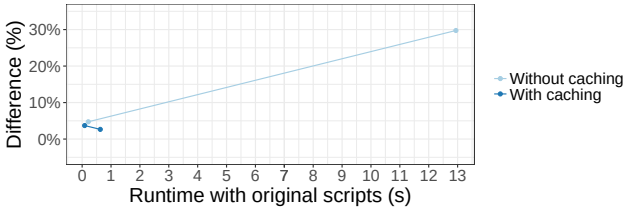


Fig. 10: Relative differences between replications with *ExpRunA* and with original scripts for Experiment 2

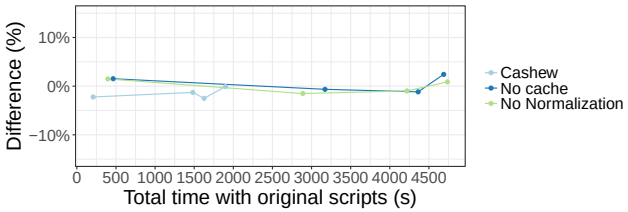


Fig. 11: Relative differences between replications with *ExpRunA* and with original scripts for Experiment 3

for Obscure regarding the number of hits and misses. This discrepancy is likely due to changes in the version of ABC [3] constraint solver. Nevertheless, a further investigation should be carried out to confirm or refute this hypothesis. The relative differences between the total runtime results with *ExpRunA* and with original scripts are presented in Figure 11. The differences are below 5%, for all cases.

5.2 Level of abstraction

Besides execution and analysis automation, we evaluated *ExpRunA* with respect to the level of abstraction of experiment specifications from the perspective of experimenters. The assessment of the level of abstraction is guided by the following research questions:

RQ 4. Can *ToExpDSL* raise the level of abstraction of technology-oriented experiment specifications?

RQ 5. What is the level of abstraction of *ToExpDSL*'s language constructs?

5.2.1 Evaluation method

To address RQ 4, we compared the level of abstraction of the original specifications with specifications using *ToExpDSL* for the experiments used in Section 5.1. The level of abstraction is evaluated based on the following criteria:

- **Level of detail:** abstract specifications say what a program does without necessarily saying how it is done; abstraction is a process of generalization, eliminating detail, removing inessential information [51].
- **Number of potential implementations:** abstract specifications have more potential implementations, whereas moving to a lower level means restricting the number of potential implementations [51].
- **Complexity:** abstract specifications are less complex than low-level specifications [27].
- **Domain concepts:** abstract specifications use concepts related to their domain of application [27].

To address RQ 5, we first selected well-established guides in Software Engineering experimentation and then compared their key concepts with *ToExpDSL*'s constructs/elements. Then, we classified the *ToExpDSL*'s constructs in three groups, according to their relation with domain concepts:

- **High-level construct:** a construct that is directly related to a domain concept found in the literature.
- **Mid-level construct:** a construct that is not directly related to a domain concept but supports or details high-level constructs.
- **Low-level construct:** a construct that neither is directly related to a domain concept nor supports or details high-level constructs.

5.2.2 Results

We present the results of comparing the level of abstraction of the original specifications with specifications written in our *ToExpDSL*, which we used in Section 5.1 and also the results of comparing the *ToExpDSL*'s constructs/elements with key experimentation concepts.

Experiment specifications

For execution, the authors of Experiment 1 created an execution script (Listing 1) and a Gnuplot configuration file (Listing 3). Instead, we created a corresponding specification using *ToExpDSL* (Listing 14). We defined the treatments (Lines 11–14) and associated the parameter `params` to each treatment. Likewise, we defined the objects (Lines 15–19) and associated the parameters `num_steps` and `step_size` to them. Section `Executions` (Lines 20–27) defines the commands to be executed. We defined the main command (Line 22), i.e., the command which is measured, and a preprocessing command (Line 24). The preprocessing command is executed before the main command as a preparatory step and is not measured. The parameters associated with the treatments and objects are used both in the main command (`object.parameter.num_steps`) and in the preprocessing command (`object.parameter.num_steps`,

Table 5: Results of SPF-based quantitative analyses of string programs [12]

Program	Caching	Total time	Speedup	#Hits	#Misses	H/M
Password1	None	297 s	–	–	–	–
	No norm	258 s	1.15x	17,547	56,173	0,31
	Cashew	106 s	2.80x	62,797	10,923	5.75
Password2	None	3,364 s	-	-	–	–
	No norm	3,379 s	0.99x	30,448	824,832	0.04
	Cashew	1,243 s	2.71x	659,804	195,476	3.38
Obscure	None	2,158 s	–	–	–	–
	No norm	1,965 s	1.10x	2,000	59,000	0.03
	Cashew	609 s	3.54x	44,893	16,107	2,79
CRIME	None	3,005 s	–	–	–	–
	No norm	2,941 s	1.02x	31,884	84,127	0.38
	Cashew	1,067 s	2.82x	78,289	37,722	2.08

Table 6: Results of SPF-based quantitative analyses of string programs (original scripts)

Program	Caching	Total time	Speedup	#Hits	#Misses	H/M
Password1	None	463.61 s	–	–	–	–
	No norm	395.78 s	1.17x	17,547	56,173	0,31
	Cashew	208.51 s	2.22x	62,797	10,923	5.75
Password2	None	4,689.48 s	–	–	–	–
	No norm	4,737.73 s	0.99x	30,448	824,832	0.04
	Cashew	1,899.93 s	2.47x	659,804	195,476	3.38
Obscure	None	3,172.23 s	–	–	–	–
	No norm	2,888.71 s	1.10x	1,999	58,999	0.03
	Cashew	1,482.03 s	2.14x	32,443	28,555	2,79
CRIME	None	4,362.45 s	–	–	–	–
	No norm	4,218.28 s	1.03x	31,884	84,127	0.38
	Cashew	1,626.53 s	2.68x	78,289	37,722	2.08

Table 7: Results of SPF-based quantitative analyses of string programs (*ExpRunA*)

Program	Caching	Total time	Speedup	#Hits	#Misses	H/M
Password1	None	470.68 s	–	–	–	–
	No norm	401.66 s	1.17x	17,547	56,173	0,31
	Cashew	203.87 s	2.31x	62,797	10,923	5.75
Password2	None	4,803.33 s	–	–	–	–
	No norm	4,779.28 s	1.01x	30,448	824,832	0.04
	Cashew	1,898.29 s	2.53x	659,804	195,476	3.38
Obscure	None	3,151.55 s	–	–	–	–
	No norm	2,844.71 s	1.11x	1,999	58,999	0.03
	Cashew	1,462.96 s	2.15x	32,443	28,555	2,79
CRIME	None	4,311.08 s	–	–	–	–
	No norm	4,176.90 s	1.03x	31,884	84,127	0.38
	Cashew	1,5856.8 s	2.72x	78,289	37,722	2.08

object.parameter.step_size, and treatment.parameter.params) to apply the corresponding treatment to the object.

Listing 15 presents an excerpt of the specification of Experiment 2. We defined the treatments (Lines 6–9) and associated the parameter conf to each treatment. We also defined the objects (Lines 10–14) and associated the parameter prefix to them. We used both treatments and objects parameters in the command definition (Line 17).

Listing 16 presents an excerpt of the specification of Experiment 3. In this case, we used the treatment name (treatment.name) and the object name (object.name) to define the command (Line 21).

The complete execution scripts and experiment specifications of Experiments 1 to 3 are presented on our supplementary Website (http://expruna.github.io).

Based on the comparison criteria defined in Section 5.2.1, we compared the level of abstraction of the original specifications with specifications written in *To-*

Listing 14: Excerpt of the specification of Experiment 1

```
Experiment hylaaOptimization {
  Research Hypotheses {
    RH1 {time Hylaa = Warm description "Runtime time for Hylaa is equal to runtime time for Warm" }
  }
  Experimental Design {
    runs 10
  }
  Dependent Variables {
    time { description "Runtime" scaleType Absolute unit "seconds" instrument timeInstrument }
  }
  Treatments {
    Hylaa description "Hylaa" factor optimization parameters {params ""} execution hylaaTool,
    Warm description "Warm" factor optimization parameters {params "settings.opt_decompose_lp=False"} execution hylaaTool
  }
  Objects {description "Number of steps" {
      steps31 {description "31 steps" value "31" parameters {num_steps "31", step_size "0.200000000"}},
      steps106948 {description "106948 steps" value "106948" parameters {num_steps "106948", step_size "0.000058720"}}
    }
  }
  Executions {
    hylaaTool {
      command "/usr/bin/python -u hybridpy/tool_hylaa.pyc ${treatment.name}/${object.parameter.num_steps}.py -"
      preprocessing {
        hyst{command "java -jar /opt/hyst-1.5/src/Hyst.jar -i /opt/optimizations/io.xml -o
            /opt/models/${treatment.name}/${object.parameter.num_steps}.py -tool hylaa '-settings
            settings.print_output=False ${treatment.parameter.params} -step ${object.parameter.step_size}'"}
    }
   }
}
```

Listing 15: Excerpt of the specification of Experiment 2

```
Experiment cashew {
  description "Constraint Normalization and Parameterized Caching for Quantitative Program Analysis"
  Research Hypotheses {
    RH1 {averageTime cashew = noCache description "Average time for Cashew is equals than Average time for No Cache"}
  }
  Treatments {
    cashew description "All transformations" parameters{conf "kaluza.cashew.conf"} execution cashewExecutor},
    noCache description "No cache" factor transformations parameters{conf "kaluza.nocache.conf"} execution cashewExecutor
}
  Objects {description "Constraints" {
    small {description "SMC-Small" parameters {prefix "small"}},
    big {description "SMC-Big" parameters {prefix "big"}}
    }
  }
  Executions {
    cashewExecutor {
      command "run-orbits.sh ${treatment.parameter.conf} ${object.parameter.prefix}"
    }
  }
}
```

Listing 16: Excerpt of the specification of Experiment 3

```
1  Experiment cashew {
2    description "Constraint Normalization and Parameterized Caching for Quantitative Program Analysis"
3    Research Hypotheses {
4      RH1 {sumTime cashew = nocache description "Total time for Cashew is equals than Total time for No Cache"},
5      RH2 {sumTime cashew = trivialcaching description "Total time for Cashew is equals than Total time for No Normalization"}}
6    }
7    Treatments {
8      nocache description "No cache" parameters{conf "kaluza.nocache.conf"} execution cashewExecutor,
9      trivialcaching description "No Normalization" parameters{conf "kaluza.cashew-except-order.conf"} execution cashewExecutor,
10     cashew description "Cashew" parameters{conf "kaluza.cashew.conf"} execution cashewExecutor
11   }
12   Objects { description "Constraints"{
13       password {description "Password1"},
14       password2 {description "Password2"},
15       obscure {description "Obscure"},
16       crime {description "CRIME"}
17     }
18   }
19   Executions {
20     cashewExecutor {
21     command "/root/phab/jpf-security/src/examples/cashew/get-results-security.sh ${treatment.name} ${object.name}"
22   }
23 }
```

ExpDSL, which we used in Section 5.1, thus obtaining the following results:

- **Level of detail:** Since *ToExpDSL* is declarative, it only says what the experiment does without saying how to do it. The details of how to execute and analyze an experiment are specified in the code generators. Using Python, an experimenter must write how to execute and analyze the experiment with all implementation details. For instance, using *ToExpDSL*, an experimenter needs only to specify the experimental design (Listing 14, Lines 5–7), treatments (Lines 11–14), and objects (Lines 15–19). The details of how to apply the treatments to the objects are implemented in the code generators, according to the experimental design. On the other hand, using Python (Listing 1), one must write not only the treatments and objects definitions but also the details of applying the treatments to the objects. For example, two loops (Lines 13 and 18) are used to apply treatments to objects.
- **Number of potential implementations:** *ToExpDSL* is implemented by the specific code generators, which are able to generate any text. So, code generators can generate source-code in any other language, including another DSL. To provide a distinct implementation of the execution script in Python, one would have to use distinct implementations of the Python compiler, which, indeed, limits the potential implementations.
- **Complexity:** Since *ToExpDSL* is declarative, it does not contain control flow statements. All the complexity is left to the code generators, which, once created, do not need to be directly used by experimenters. To create corresponding execution scripts in Python, or any other imperative language, the experimenter must deal with the complexity of control flow statements, variable declarations, unsuitable state, and so on. For instance, in Listing 1, to repeat the execution a number of times, first, the variable num_trials is declared (Line 3). Then, a loop control is used to repeat the execution the number of times defined in num_trials (Lines 21–25). Using *ToExpDSL*, the experimenter simply defines the number of runs (Listing 14, Line 6).
- **Domain concepts:** *ToExpDSL* was created to be used for technology-oriented experimenters. So, naturally, it uses domain concepts, such as Research Hypothesis (Listing 14, Line 2), Dependent Variables (Line 8), Treatments (Line 11), Objects (Line 15), etc. Unlike *ToExpDSL*, the original scripts were created using Python, which is a general purpose language and does not contain any concept related to experiments such as those mentioned previously in this bullet item. Instead, as mentioned in the previous bullet item, such general purpose language has low-level constructs such as control flow statements and variable declarations, which obfuscate understanding at the domain level.

DSL Constructs

We also present a comparison between *ToExpDSL*'s constructs and domain concepts. In this comparison, we considered all types defined in *ToExpDSL*'s grammar. Based on the criteria defined in Section 5.2.1, we classified the grammar constructs in three groups: high-level constructs (Table 8), mid-level constructs (Table 9), and low-level constructs (Table 10).

As a result of the evaluation of the level of abstraction of *ToExpDSL*'s language constructs (RQ 5), we found that, out of 46 types defined in the grammar, 25 are high-level constructs (54.35%), 7 are mid-level constructs (15.22%), and 14 are low-level constructs (30.43%).

5.3 Analysis and interpretation

In this section, we analyze and interpret the results from the previous subsections in terms of the corresponding research questions. We also discuss threats to validity, limitations of the contribution, and future work.

The results from Section 5.1.3 provide evidence that *ExpRunA* can be used to specify different technology-oriented experiments (RQ 1). First, the experiment specifications resulting from the replications suggest that *ExpRunA* supports the researcher in specifying an experiment by providing a specific editor with syntax highlighting, content assist, syntax validation, static semantics validation, template proposals, and text hover.

Second, when it comes to the execution results, they suggest that not only *ExpRunA* facilitates automation of execution and analysis from the specification of technology-oriented experiments but also that the generated execution and analysis scripts are sound (RQs 2 and 3). Indeed, by design, from experiment specifications, *ExpRunA* aims at providing a push-button solution that automatically generates execution and analysis scripts, runs the execution script, analyzes the results, and presents the analysis results to the researcher from an experiment specification at a high-level of abstraction. In particular, the running infrastructure (Dohko) runs the execution script, reports the execution status, and provides execution results; the analysis infrastructure (R Sweave environment) analyzes the execution results and generates an analysis report.

Although there are some differences regarding execution time between the replications with *ExpRunA* and with original scripts, the qualitative results are consistent and lead to the same conclusions. These results suggest that the overhead of *ExpRunA* is more significant for lower runtime values (below one second), although the results for Experiment 2 diverge from this hypothesis.

Regarding abstraction level (Section 5.2.2), we conclude that *ExpRunA* raises the level of abstraction required to execute and analyze a technology-oriented experiment (RQ 4), since the use of *ToExpDSL* empowers researchers to specify experiments using experimentation concepts (e.g., experimental design, treatment, experimental object, dependent variable). A model-driven approach is then used to generate execution and analysis scripts from the experiment specification. Since code generators generate execution and analysis scripts, this frees the researcher from dealing with the low-level details of creating such scripts.

Moreover, regarding RQ 5, despite the existence of some low-level constructs, the high-level and mid-level constructs add up to around 70%. In addition, the low-level constructs are not too complex since they are declarative statements instead of control flow statements. All the low-level constructs are related to the infrastructure, which suggests that these constructs should not be part of *ToExpDSL*. Instead, they should be defined somewhere in the supporting framework. Furthermore, this also suggests that there is another role in the experimentation process, a system administrator, which deals with low-level details to configure the required infrastructure to run the experiment. In fact, special attention must be payed to the *Requirements* construct. Although this construct reflects infrastructure requirements, such as CPU, memory, and costs, in some experiments, these specifications are important for the context of the experiment. Thus, there should be a way to specify these requirements using the Context construct, and have the code generators map then to the infrastructure requirements. By doing so, the number of high-level constructs would increase to 78.13%, the number of mid-level to 21.88%, and there would not be low-level constructs anymore.

5.3.1 Threats to validity

The evaluation of automation (Section 5.1) is a quantitative evaluation based on replications. On the other hand, the evaluation of the level of abstraction (Section 5.2) is an analytical comparison. For both evaluations, we present the threats to validity:

Conclusion validity To perform the replications with *ExpRunA* and with original scripts, we used procedures and scripts as similar as possible to those of the original papers. This includes the number of runs, which affects the sample size, and the procedure to collect execution results. For this reason, we could not perform statistical significance tests to check the differences in re-

Table 8: Comparison between *ToExpDSL*'s constructs and Domain Concepts (high-level constructs)

ToExpDSL Construct	Domain Concept		
	Jedlitschka et al. [25]	Wohlin et al. [53]	Juristo and Moreno [26]
Abstract	Abstract	Abstract	N/A
Analysis	Analysis	Data Analysis	Analysis
Author	Authorship	Authorship	N/A
Context	Parameter	Context	Parameter
DependentVariable	Dependent variable	Dependent variable	Response variable
DesignType	Design Type	Design Type	Design Type
Execution	Execution	Execution	Execution
Experiment	Experiment	Experiment	Experiment
ExperimentalDesign	Experiment Design	Experiment design	Experimental design
ExperimentalObject	Experimental Material	Object	Experimental object
Factor	Independent variable	Factor	Factor
Goal	Goal	Goal	Goal
Instrument	Instrument	Instrument	N/A
Keyword	Keyword	N/A	N/A
Range	Range	Range	N/A
ResearchHypothesis	Hypothesis	Hypothesis	Hypothesis
ResearchQuestion	Research question	Research question	N/A
ScaleType	Scale type	Scale type	Scale type
SimpleAbstract	Abstract	Abstract	N/A
SimpleGoal	Goal	Goal	Goal
StructuredAbstract	Structured Abstract	Structured Abstract	N/A
StructuredGoal	Goal	Goal	Goal
Threat	Threat to validity	Threat to validity	Validity threat
ThreatType	Threats classification	Threats classification	Threats classification
Treatment	Treatment	Treatment	Level

Table 9: DSL mid-level constructs

ToExpDSL Construct	Purpose
File	Related to a Treatment or to an Experimental Object
Model	Container for all elements of the grammar
ObjectGroup	Groups related Experimental Objects
OperatorType	Represents which comparison between Treatments will be done
Parameter	Related to a Treatment or to an Experimental Object
ResearchHypothesisFormula	Comprises a Dependent Variable, two Treatments, and an Operator Type
Restriction	Used to limit the relation between Treatments and Experimental Objects

Table 10: DSL low-level constructs

ToExpDSL Construct	Purpose
AccessKey	Cloud Access Key
Infrastructure	Infrastructure specifications
InstanceType	Virtual Machine Instance Type
Cloud	Cloud specifications
CloudProvider	Cloud Provider specification
OnFinishType	Action performed in the virtual machine after finishing execution
PlatformType	Virtual Machine Platform Type
Preconditions	Names of packages required to run the experiment
Region	Cloud Region
Requirements	Infrastructure requirements, such as CPU, memory, cost, etc
StatusType	Region Status
User	Username and User Keys
UserKey	User key to access the Cloud
Zone	Cloud Zone

sults between the executions with and without our tool. In Experiment 1, each treatment was applied to each object ten times; however, the original script records only the mean, the minimum, and the maximum value of each sample, which is not sufficient information to perform a significance test. This would require all the single measurements, or, at least, the mean and the variance of the sample [11, 26]. In Experiments 2 and 3, since each object is, in fact, a whole dataset, each treatment was applied only one time to each object, which results in an insufficient sample size to perform a significance test. Therefore, we drew our conclusions based on the interpretation of the plots containing execution results, and considering the qualitative results of each replication. To mitigate the threat of using a bad instrumentation, in the replications, we used the same instrumentation as used in the original experiments.

Internal validity The measurement of performance, specially runtime, is sensibly affected by the execution environment. Other processes running on the same machine and consuming resources, such as CPU, memory, and disk access, may cause variations in runtime. This could affect the comparison of the results of replications with *ToExpDSL* and with original scripts. To reduce this threat, we ran each replication in a dedicated virtual machine on Google Cloud. The virtual machine was recreated before each replication using the same configuration to keep the execution environments as similar as possible.

Construct validity To assure that the metrics chosen for the evaluation are suitable measures of the issue under investigation, they were derived from the goals and research questions and based on the literature. RQs 1 to 3 assess the ability of *ExpRunA* to provide automation in the experimentation process. Since it is a feasibility evaluation, we evaluated if we can use *ToExpDSL* and *ExpRunA* to run previously published experiments. To assure that *ExpRunA* produces sound results, we also compared their results with results produced by original scripts. To assess RQ 4 and RQ 5, we based our comparison on criteria derived from the literature concepts.

External validity To empirically evaluate our solution, we replicated distinct experiments from the automatic verification domain. To find technology-oriented experiments completely documented and with all the artifacts available (Section 5.1.2), we directed our search to publication venues explicitly requiring reproducibility as part of the evaluation process or highlighting this fact in accepted papers. In future works, we intend to replicate experiments from additional domains and also compare the level of abstraction of these experiment specifications with experiment specifications using our tool.

Reliability We conducted the evaluation ourselves, which way introduce bias. In relation to automation, since it is a feasibility evaluation, and not a subjective evaluation, such as usability, the bias does not affect the results. When it comes to the evaluation of abstraction, to mitigate the threat of researchers bias, we defined objective evaluation criteria based on references from the literature.

5.3.2 Limitations and future work

Although *ToExpDSL* is sufficiently expressive to specify technology-oriented experiments, and *ExpRunA* can be used to enable automation of execution and analysis of technology-oriented experiments, there are some limitations.

Experiment design The experiment design relies on a (subset of) Cartesian product to relate treatments and experiment objects. There should be a means to specify additional designs relating more than two treatments at a time, or even applying only one treatment to several objects. In future work, we plan to support additional design types relating more than two treatments at a time, or applying only one treatment to several objects in scalability evaluations since, currently, we support only two-treatment comparisons. For example, Lanna et al. [30], besides the comparison among analysis strategies, they evaluated their scalability.

Experiment objects *ExpRunA* is able to apply a treatment to an object multiple times as defined by the experimenter. However, the object must be exactly the same. In some experiments [10, 17], the treatment is applied to a group of related objects, and all the measurements are analyzed as if they were repetitions of the same object. In future work, we plan to support the definition of groups of related objects so that, for each group, the results can be analyzed as if they were repetitions of the same object. Additionally, *ExpRunA* does not currently support hierarchically defined objects with parameterization at any level of depth of the hierarchy, which can be provided, for instance, by the cyber communication network simulation tool OMNeT++ [48]. We also envision this enhancement as future work.

Output checking Using *ExpRunA*, the applications corresponding to the treatments are executed and the dependent variables are measured. However, there is no way to compare the output of the tool with some reference value. This would be necessary, for instance, to replicate the experiment presented in Beyer et al. [10]. In future work, we plan to provide means to specify an output reference to check the actual output of execution,

which would assure that the tool related to the treatment is performing the work it is supposed to do rather than performing some arbitrary processing.

Dynamic task definition Although some design studies demand dynamic task definition, *ExpRunA* currently does not support this feature. Instead, it can be extended to support the concepts of block (Listing 12, Section 4.3), block dependency, and command currently supported by the underlying infrastructure (Section 4.2) but not exposed at the *ToExpDSL* level yet, to enable users to check whether a given condition is met at runtime. Nevertheless, users have to explicitly define all the tasks statically following a declarative strategy, which still offers a valuable aid in different domains such as fluid simulations and bioinformatics. In future work, we plan to evolve our tool to handle on-the-fly tasks.

Analysis Since the research hypotheses relate only two treatments, the statistical tests performed are T test and Mann–Whitney, depending on normality of the data. We plan to provide additional statistical tests and allow the experimenter to choose the tests to be applied and plots to be generated. Supporting additional designs also means providing additional tests corresponding to these designs. In addition, the experimenter should have more control over the statistical tests being applied and the plots being generated. In future work, we plan to abstract the type of analysis into the DSL level.

Evaluation Our evaluation results suggest that the differences in runtime with *ExpRunA* and with original scripts vary for distinct time ranges. However, this should be thoroughly investigated with additional experiments. In future work, we plan to replicate the same experiments again but changing the original scripts so that we can collect enough data to perform significance tests to compare the results with and without the tool. In addition, we evaluated neither the cost of learning *ToExpDSL* nor its usability, which we plan to do in future work. This would provide more information regarding the costs of the adoption of our solution to experimenters who want to use it to conduct their experiments.

Manual tasks Although *ExpRunA* facilitates the automation of execution and analysis of experiments, the experimenters still have to interpret the results, draw the conclusions, write replication instructions, and publish the lab package. In addition to these manual tasks, a system administrator has to properly configure the running infrastructure to run the experiment, for instance, installing specific tools, packages, and dependencies on it. Then, the system administrator can publish a Docker image with these configurations so that other researchers can replicate the experiment or conduct further analyses. Further details and examples are presented on our supplementary Website. In future work, we plan to relate some *ToExpDSL*'s constructs to the role of system administrator and others to the role of experimenter to achieve a better separation of roles.

Execution Optimization. *ExpRunA* relies on an extensional specification provided by the experimenter. This means that, when the number of combinations of treatments and objects increases, the experimenter is responsible for explicitly specifying the relevant subset of the Cartesian product to avoid unnecessary executions. To cope with this potential state explosion, a future improvement could be to use a constraint language to help deriving these subsets automatically. While a constraint language would not reduce the semantic problem complexity, it would allow an intentional specification thus reducing the burden on the user. Another option would be relying on algorithms to explore the parameter spaces and to generate the tasks to execute on-the-fly automatically, as supported by OpenTURNS [8] and OpenMETA [44].

Distributed execution We evaluated our solution considering a local and a non-distributed execution scenario. In the case of a large combination of executions, the executions will be scheduled sequentially, which can lead to a large amount execution time. As a future enhancement, we plan to use Dohko to execute the applications in a distributed way on the cloud, which is, in fact, already supported by Dohko.

Despite the limitations discussed previously, we argue that the scope of experiments currently addressed is practically relevant, so explicitly acknowledging and discussing these limitations provides a natural and useful path for further development. Indeed, during our research, we found several experiments that fit to the scope but could not be used in our evaluation due to reproducibility issues, such as missing documentation or artifacts [16, 41], artifact compilation problems [13], or the need of a large amount of resources to run the experiment [1].

6 Related work

To address the problems related to conducting experiments, many techniques have been proposed. To the best of our knowledge, none of them simultaneously addresses the executable specification of experiments at a high level of abstraction; automated treatment execution and automated data analysis from the experiment specification; and formal guaranties of the correctness of results with respect to the experiment specification for technology-oriented experiments. The existing techniques have a different and broad perspective and support distinct phases of the experimentation process

either for technology-oriented or for human-oriented experiments.

Technology-oriented experiments Beyer et al. [9] formulated a set of requirements for reliable benchmarking and accurate resource measurements. They also provided *BenchExec*, a free implementation of a benchmarking framework that fulfills all presented requirements. The authors first defined some restrictions of the tool to be run: the tool is CPU-bound, i.e., when compared to CPU usage, input and output operations from and to disks are negligible, and input and output bandwidth does not need to be limited nor measured; the tool does not perform network communication during the execution; the tool does not spread across several machines during execution, but is limited to a single machine; and the tool does not require user interaction. Based on these restrictions, the author listed five specific requirements for reliable benchmarking: measure and limit resources accurately, kill processes reliably, assign cores deliberately, respect non-uniform memory access, and avoid swapping. Then, the authors described *BenchExec*, a cgroups-based benchmarking framework that fulfills all these requirements. *BenchExec* is split in two parts, one responsible for benchmarking a single run of a given tool, named *runexec*, and the other responsible for benchmarking a whole set of runs. The tool *runexec* can be easily used from within other benchmarking frameworks. In fact, we integrated *runexec* with *Dohko* [31] so that our execution environment meets the requirements presented by the authors.

Hauck et al. [22] presented Goal-oriented INfrastructure Performance EXperiments (Ginpex) approach, which introduces goal-oriented and model-based specification and generation of executable performance experiments for automatically detecting and quantifying performance-relevant infrastructure properties. Ginpex provides a meta-model for experiment specification and comes with predefined experiment templates that provide automated experiment execution on the target platform and also automate the evaluation of the experiment results. It can be used by performance analysts to automatically derive performance-relevant infrastructure properties for performance predictions. Like our approach, Ginpex provides automated execution and data analysis. However, the main focus of Ginpex is to derive performance-relevant infrastructure properties based on goal-oriented measurements, whereas our work focuses on technology-oriented experiment description guided by research hypotheses, automated execution, and data analysis. Ginpex's evaluation consisted of designing experiments for detecting time slice length of operating system schedulers, and for quantifying performance penalty of virtualized computing resources through Xen hypervisor [7]. Ginpex could be used, for instance, to evaluate the overhead of *ExpRunA* and the performance characteristics of the running infrastructure.

Wang et al. [50] presented Weevil, a framework providing techniques for software engineers to automate the experimentation activity in highly distributed systems. A highly distributed system usually consists of a network of components, executing independent and possibly heterogeneous tasks, that collectively realize a coherent service. Their approach is founded on a suite of models that characterize the distributed system under experimentation, the testbeds upon which the experiments are to be carried out, and the client behaviors that drive the experiments. Similar to our approach, Weevil uses a model-based approach to provide automated execution from an experiment configuration. However, it does not provide automated data analysis from the experiment specification. In addition, its main focus is on highly distributed systems.

Human-oriented experiments Freire et al. [20] proposed a model-driven approach to specify and monitor controlled experiments in software engineering, focusing on human-oriented experiments. Their approach comprises a DSL, named ExpDSL; model-driven transformations that allow workflow models generation; and a workflow execution environment. First, a researcher uses ExpDSL to specify the experiment. Then, model-driven transformations are applied to the experiment specification to generate customized workflows for each experiment participant. Finally, the workflow is executed in a Web-based workflow engine, which guides the participants by providing instructions for their tasks. In addition, the researchers running the experiment can monitor the activities performed by the participants. Their approach is similar to ours in the sense that they use a DSM approach comprising a DSL, code generators, a supporting framework, and a running infrastructure. However, there are significant differences. First, unlike our approach, their work supports human-oriented experiments. For this reason, we partially based our DSL in ExpDSL but we extended it with new constructs for technology-oriented experiments. Second, their approach does not provide data analysis. Finally, since we enable automation of execution and data analysis of technology-oriented experiments, our code generators, supporting framework, and running infrastructure are completely different. Although their approach can be used for scoping, planning, and execution, it is not suitable for technology-oriented experiments.

Travassos et al. [46] presented an experimental Software Engineering Environment (eSEE) to support large-scale experimentation and scientific knowledge manage-

ment in Software Engineering. It is represented by a computerized infrastructure to support large-scale experimentation in Software Engineering. eSEE provides a set of facilities to allow geographically distributed software engineers and researchers to accomplish and manage experimentation processes as well as scientific knowledge concerned with different study types through the web. The eSEE's conceptual model has been organized in three abstraction levels: meta, configured, and execution. Meta-level contains common knowledge regarding experimental software engineering and its studies, including Software Engineering knowledge. Configured-level is the knowledge for each type of experimental study. Finally, execution-level is the knowledge for a specific study, which is the only level supported by *ExpRunA*. Further, their proposal includes definition, planning, execution, and packaging of primary and secondary studies. However, unlike *ExpRunA*, eSEE does not support automated execution and data analysis from the experiment specification for technology-oriented experiments. A possible future synergy development between eSEE and *ExpRunA* would be to integrate eSEE's knowledge base into our tool and endow the latter with meta-analyses capability thus further raising the evidence of the conducted experiments.

Arisholm et al. [2] developed a Web-based experiment support environment called Simula Experiment Support Environment (SESE) to support large-scale human-oriented experiments. The objective is to scale up the experiments and particularly run experiments with professionals in industry using professional development tools to make the experiments more realistic. SESE supports the logistics of a large-scale experiment and allows an experimenter to define experiments, including all the detailed questionnaires, task descriptions and necessary code, assign subjects to a given experiment session, run and monitor each experiment session and collect the results from each subject for analyses. However, SESE is bound to human-oriented experiments and does not include data analysis.

Hochstein et al. [23] described the Experiment Manager Framework, an environment that simplifies the process of collecting, managing, and sanitizing data from classroom experiments, while minimizing disruption to natural subject behavior. The framework is an integrated set of tools to support software engineering experiments in High Performance Computing (HPC) classroom environments. The objectives are to simplify the process of conducting software engineering experiments that involve development effort and workflow, and to ensure consistency in data collection across experiments in classroom environments. The framework also supports data analysis. Some of these analyses are focused on a single subject, while others aggregate data over several classes. However, the framework does not support technology-oriented experiments.

Feigenspan et al. [19] presented a tool called PROPHET to support program-comprehension experiments. Their aim is to provide a tool infrastructure that can be used and extended by other researchers to design and conduct program-comprehension experiments. PROPHET is highly customizable, and its plug-in structure allows researchers to implement new features without changing the existing source code of PROPHET. However, it does not provide analysis, and program-comprehension experiments are inherently human-oriented experiments.

Scientific workflows Scientific workflows are used to model a flow of activities and data ready to be executed by a workflow engine. Scientific workflows are an alternative to represent pipelines or script-based applications. In scientific workflows, these activities are often programs or services that represent solid algorithms and computational methods [35]. The purpose of our approach is not to replace scripts or scientific workflows; instead, it is to generate scripts from high-level experiment specifications. The sequence of activities to be executed by the scripts is derived from experimentation concepts, such as research hypotheses, treatments, objects, dependent variables, and experimental design. Likewise, we could use our approach to generate a workflow model from the experiment specification by creating specific code generators and replacing the running infrastructure by a workflow engine.

Data analysis and presentation Madeyski and Kitchenham [33] discussed the concept of *Reproducible Research* and its use to address some problems found in empirical software engineering research, particularly issues related to validity and reproduction of data analysis. The authors raised awareness of the problems caused by unreproducible research in software engineering, which is caused by a lack of raw data, sufficient summary statistics, or undefined analysis procedures. *Reproducible Research* refers to the extent to which the report of a specific scientific study can be compiled from the reported text, data, and analysis procedures. *Reproducible Research* is proposed as one of the methods to address problems with empirical research in software engineering. The authors suggested the use of a set of free and open-source tools to use in practice to produce reproducible research, including R, LaTeX, and Sweave. To avoid the issues discussed by the authors, we followed their recommendations and used R, LaTeX, and Sweave in data analysis and results presentation. In addition, the generated analysis scripts, as well the raw data and the results, become available to the experimenter.

As mentioned previously, although the aforementioned techniques help in conducting controlled experiments, they have a different and broad perspective and can be seen as complementary works.

7 Conclusion

We presented a domain-specific approach for specifying, executing, and analyzing technology-oriented experiments. The solution comprises a DSL (*ToExpDSL*), execution and analysis script generators, and a supporting framework integrating the previous components as part of a tool named *ExpRunA*. All these components are integrated in a Web-based tool (*ExpRunA*), providing a means to specify executable specifications at a high level of abstraction, automated execution, data analysis, and results presentation.

We empirically evaluated the practical applicability of *ExpRunA* to facilitate automation in the experimentation process and to raise the level of abstraction of specifying and reasoning about experiments. Our results suggest that *ToExpDSL* is sufficiently expressive to specify technology-oriented experiments and that *ExpRunA* can be used to enable sound automation of execution and analysis from the specification of technology-oriented experiments. In addition, our empirical assessment suggests that the use of *ToExpDSL* raises the level of abstraction of experiment specifications when compared to general purpose languages. However, even the low-level constructs are less complex than general purpose language statements since they contain only declarative statements instead of control flow ones.

We also devised a formal model of the approach and some key correctness properties. These correctness properties were formally proved, which assures that the results are consistent with the experiment specification.

Although the evaluation conducted in Section 5 consists of a replication of three experiments, the description of the approach itself, which is done earlier in Section 3, is not limited to these specific problems. Indeed, ToExpDSL embeds various concepts of technology-oriented experiments, and ExpRunA supports its execution without being tied to those three experiments. As acknowledged in Section 5.3, under heading External Validity, the evaluation is currently limited to these three cases, though, and we plan to extend it in the future. In fact, in most of the related work, the authors used up to three experiments to evaluate their proposal. We also point out that the correctness properties proved characterize a formal assessment, which is not limited to the three cases presented. Overall, the empirical and the formal assessment suggest that *ToExpDSL* and *ExpRunA* are a step toward improved efficiency of the experimentation process and correctness of its results.

Our DSM solution has a number limitations, which we plan to address in future work. Specifically, we plan to support additional design types, since we currently support only comparisons between two treatments. We also plan to provide additional statistical tests and support the definition of groups of related objects so that, for each group, the results can be analyzed as if they were repetitions of the same object. Moreover, we want to provide means to specify an output reference to check the actual output of execution. Furthermore, we want to conduct further experiments to investigate the overhead of the tool in relation to runtime. Finally, we want to evaluate additional aspects of *ToExpDSL*, such as usability and the cost of learning the language by independent users.

Acknowledgements We would like to thank the following people for fruitful discussions and suggestions on how to improve this work: Andre Lanna, Thiago Castro, Thiago Ramos, Alba Melo, Eduardo Nakano, Rodrigo Ribeiro, Guilherme Travassos, Rodrigo Bonifacio, Stefan Ganser, Pierre-Yves Schobbens, Gilles Perrouin, and the anonymous reviewers. Vander Alves was partially supported by CNPq (grant 310757/2018-5), CAPES (Edital 29/2017-CAPES/WBI), and the Alexander von Humboldt Foundation. Eneias Silva was partially supported by FAPDF (Edital 1/2017). Apel's work has been funded by the German Research Foundation (AP 206/11).

References

1. Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of digital controllers for state-space physical plants. In *International Conference on Computer Aided Verification*, pages 462–482. Springer, 2017.
2. Erik Arisholm, Dag IK Sjøberg, Gunnar J Carelius, and Yngve Lindsjørn. Sese an experiment support environment for evaluating software engineering technologies. In *Tenth Nordic Workshop on Programming and Software Development Tools and Techniques*, pages 81–98, 2002.
3. Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, pages 255–272, 2015.
4. Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *International Conference on Computer Aided Verification*, pages 401–420. Springer, 2017.
5. Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S Păsăreanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 193–204, 2016.
6. Jerry Banks. Introduction to simulation. In *1999 Winter Simulation Conference*, pages 7–13, 1999.
7. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and

Andrew Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.
8. Michaël Baudin, Anne Dutfoy, Bertrand Iooss, and Anne-Laure Popelin. *OpenTURNS: An Industrial Software for Uncertainty Quantification in Simulation*. Springer, 2017.
9. Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In *Model Checking Software*, pages 160–178. 2015.
10. Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on smt-based software verification. *Journal of Automated Reasoning*, 60(3):299–335, 2018.
11. George EP Box, J Stuart Hunter, and William Gordon Hunter. *Statistics for experimenters: design, innovation, and discovery*, volume 2. Wiley-Interscience New York, 2005.
12. Tegan Brennan, Nestan Tsiskaridze, Nicolás Rosner, Abdulbaki Aydin, and Tevfik Bultan. Constraint normalization and parameterized caching for quantitative program analysis. In *11th Joint Meeting on Foundations of Software Engineering*, pages 535–546, 2017.
13. Thomas Brihaye, Gilles Geeraerts, Hsi-Ming Ho, and Benjamin Monmege. M ighty l: A compositional translation from mitl to timed automata. In *International Conference on Computer Aided Verification*, pages 421–440. Springer, 2017.
14. Xiaoling Chen and Jeffrey T Chang. Planning bioinformatics workflows using an expert system. *Bioinformatics*, page btw817, 2017.
15. Marcus Ciolkowski. *An Approach for quantitative aggregation of evidence from controlled experiments in software engineering*. Fraunhofer Verlag, 2012.
16. Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Search-based similarity-driven behavioural spl testing. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, pages 89–96. ACM, 2016.
17. Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Automata language equivalence vs. simulations for model-based mutant equivalence: An empirical evaluation. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 424–429, 2017.
18. Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. 2008.
19. Janet Feigenspan, Norbert Siegmund, Andreas Hasselberg, and Markus Köppen. Prophet: Tool infrastructure to support program comprehension experiments. In *Poster at the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011.
20. Marília Freire, Paola Accioly, Gustavo Sizílio, Edmilson Campos Neto, Uirá Kulesza, Eduardo Aranha, and Paulo Borba. A model-driven approach to specifying and monitoring controlled experiments in software engineering. In *International Conference on Product Focused Software Process Improvement*, pages 65–79, 2013.
21. Marília Freire, Uirá Kulesza, Eduardo Aranha, Gustavo Nery, Daniel Costa, Andreas Jedlitschka, Edmilson Campos, Silvia T Acuña, and Marta N Gómez. Assessing and evolving a domain specific language for formalizing software engineering experiments: An empirical study. *International Journal of Software Engineering and Knowledge Engineering*, 24(10):1509–1531, 2014.
22. Michael Hauck, Michael Kuperberg, Nikolaus Huber, and Ralf Reussner. Deriving performance-relevant infrastructure properties through model-based experiments with ginpex. *Software & Systems Modeling*, 13(4):1345–1365, 2014.
23. Lorin Hochstein, Taiga Nakamura, Forrest Shull, Nico Zazworka, Victor R Basili, and Marvin V Zelkowitz. An environment for conducting families of software engineering experiments. *Advances in Computers*, 74:175–200, 2008.
24. Claudia Houben and Alexei A Lapkin. Automatic discovery and optimization of chemical processes. *Current Opinion in Chemical Engineering*, 9:1–7, 2015.
25. Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pages 201–228. 2008.
26. Natalia Juristo and Ana M Moreno. *Basics of software engineering experimentation*. Springer Science & Business Media, 2013.
27. Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.
28. Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
29. Amine Lajmi, Jabier Martinez, and Tewfik Ziadi. Dslforge: Textual modeling on the web. *DemosMoDELS*, 1255:25–29, 2014.
30. André Lanna, Thiago Castro, Vander Alves, Genaina Rodrigues, Pierre-Yves Schobbens, and Sven Apel. Feature-family-based reliability analysis of software product lines. *Information and Software Technology*, 94:59–81, 2018.
31. Alessandro Ferreira Leite, Vander Alves, Genaína Nunes Rodrigues, Claude Tadonki, Christine Eisenbeis, and Alba Cristina Magalhaes Alves de Melo. Dohko: an autonomic system for provision, configuration, and management of inter-cloud environments based on a software product line engineering method. *Cluster Computing*, pages 1–26, 2017.
32. Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *ACM SIGPLAN Notices*, volume 49, pages 565–576, 2014.
33. Lech Madeyski and Barbara Kitchenham. Would wider adoption of reproducible research be beneficial for empirical software engineering research? *Journal of Intelligent & Fuzzy Systems*, 32(2):1509–1521, 2017.
34. Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (meta) modeling: Web- and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60, 2014.
35. Marta Mattoso, Claudia Werner, Guilherme Horta Travassos, Vanessa Braganholo, Eduardo Ogasawara, Daniel Oliveira, Sergio Cruz, Wallace Martinho, and Leonardo Murta. Towards supporting the life cycle of large scale scientific experiments. *International Journal of Business Process Integration and Management*, 5(1):79–92, 2010.
36. Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *38th International Conference on Software Engineering*, pages 643–654, 2016.
37. Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.
38. SS Pavlov, A Yu Dmitriev, IA Chepurchenko, and MV Frontasyeva. Automation system for measurement of gamma-ray spectra of induced activity for multi-element high volume neutron activation analysis at the reactor ibr-2 of frank laboratory of neutron physics at the joint institute

for nuclear research. *Physics of Particles and Nuclei Letters*, 11(6):737–742, 2014.
39. Célia G Ralha, Carolina G Abreu, Cássio GC Coelho, Alexandre Zaghetto, Bruno Macchiavello, and Ricardo B Machado. A multi-agent model system for land-use change simulation. *Environmental Modelling & Software*, 42:30–46, 2013.
40. T. Rizzo, J. Duong. The crime attack. *Ekoparty Security Conference*, 2012.
41. Ana B Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 41–50. IEEE, 2014.
42. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
43. Mirko Sonntag, Dimka Karastoyanova, and Frank Leymann. The missing features of workflow systems for scientific computations. In *Software Engineering*, pages 209–216, 2010.
44. Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. OpenMETA: A model-and component-based design tool chain for cyber-physical systems. In *Joint European Conferences on Theory and Practice of Software*, pages 235–248, 2014.
45. Sajad Tabatabaei. A probabilistic neural network based approach for predicting the output power of wind turbines. *Journal of Experimental & Theoretical Artificial Intelligence*, pages 1–13, 2016.
46. Guilherme H Travassos, Paulo Sérgio Medeiros dos Santos, Paula Gomes Mian, Arilo Cláudio Dias Neto, and Jorge Biolchini. An environment to support large scale experimentation in software engineering. In *13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 193–202, 2008.
47. Guilherme Horta Travassos and Márcio O Barros. Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering. In *2nd Workshop on Empirical Software Engineering the Future of Empirical Studies in Software Engineering*, pages 117–130, 2003.
48. András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 60:1–60:10, 2008.
49. Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *SIGSOFT FSE*, 2012.
50. Yanyan Wang, Matthew J Rutherford, Antonio Carzaniga, and Alexander L Wolf. Automating experimentation on distributed testbeds. In *20th IEEE/ACM international Conference on Automated software engineering*, pages 164–173, 2005.
51. Martin Ward. A definition of abstraction. *Journal of Software: Evolution and Process*, 7(6):443–450, 1995.
52. Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *17th ACM conference on Computer and communications security*, pages 162–175, 2010.
53. Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
54. Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on JSSPP*, pages 44–60. Springer, 2003.
55. Yong Zhao, Xubo Fei, Ioan Raicu, and Shiyong Lu. Opportunities and challenges in running scientific workflows on the cloud. In *CyberC*, pages 455–462, 2011.