Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A Domain-Specific Modeling Approach Supporting Technology-oriented Experiments

Eneias Cordeiro da Silva

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A Domain-Specific Modeling Approach Supporting Technology-oriented Experiments

Eneias Cordeiro da Silva

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Coorientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2018

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

```
         Cordeiro da Silva, Eneias
CSI586d     A Domain-Specific Modeling Approach Supporting
         Technology-oriented Experiments / Eneias Cordeiro da Silva;
         orientador Vander Ramos Alves; co-orientador Alba Cristina
         Magalhães Alves de Melo. -- Brasília, 2018.
            150 p.


            Dissertação (Mestrado - Mestrado em Informática) --
         Universidade de Brasília, 2018.


            1. Experimentos controlados. 2. Experimentos orientados
         a tecnologia. 3. Modelagem específica de domínio. 4.
         Linguagem específica de domínio. I. Ramos Alves, Vander,
         orient. II. Cristina Magalhães Alves de Melo, Alba, co
         orient. III. Título.
```

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A Domain-Specific Modeling Approach Supporting Technology-oriented Experiments

Eneias Cordeiro da Silva

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Brasília, 12 de julho de 2018

Dedicatória

Dedico este trabalho ao meu pai Agenor (*in memorian*) e à minha mãe Regina. Aos meus irmãos Narciso, Dorotéia, Aline, Raquel, Jonas e Joel. À minha esposa Diesse, ao meu filho Emmanuel e à minha filha Sophia.

Agradecimentos

À Diesse, minha esposa, e ao Emmanuel, meu filho, pelo apoio incondicional e também pela paciência em compreender os momentos de ausência. Agradeço também por me incentivarem nos momentos de desânimo.

Aos meus pais Agenor (*in memorian*) e Regina, e aos meus irmãos Narciso, Dorotéia, Aline, Raquel, Jonas e Joel, por terem sido meus primeiros professores e por servirem de exemplo de dedicação aos estudos.

Ao meu orientador, Prof. Vander Alves, pelos ensinamentos e pela valiosa orientação ao longo da pesquisa, sempre me desafiando a ir mais além. Estendo também os agradecimentos à minha coorientadora Prof.ª Alba Alves de Melo pelas valorosas contribuições.

Ao Alessandro Leite, que participou intensamente desde o início do projeto com o apoio ao uso do Dohko, o que enriqueceu consideravelmente o trabalho. Agradeço também ao Prof. Rodrigo Ribeiro, pelas contribuições em relação à formalização, e ao Prof. Eduardo Nakano, pelas contribuições em relação à análise estatística.

À Fundação de Apoio à Pesquisa do Distrito Federal (FAPDF), pelo apoio na realização da visita técnica ao Departamento de Ciência da Computação e Matemática da Universidade de Passau, Alemanha. Agradeço também ao Prof. Sven Apel pela recepção durante a visita técnica e pelas preciosas contribuições ao trabalho.

Aos colegas de orientação André Lanna, Thiago Castro, Ricardo Lima e Junier Amorim, pela inspiração de seus trabalhos e pelos comentários e sugestões sempre muito úteis.

Aos membros da banca, Prof. Guilherme Travasos e Prof. Rodrigo Bonifácio, pelas sugestões e críticas que contribuíram para o aprimoramento do trabalho.

Ao Prodasen e ao Senado Federal, pela concessão de licença para capacitação, a qual possibilitou dedicação mais intensa na pesquisa durante o período, bem como a realização da visita técnica à Universidade de Passau.

"We should be taught not to wait for inspiration to start a thing. Action always generates inspiration. Inspiration seldom generates action."

(Frank Tibolt)

Resumo

Contexto: Experimentação é um meio de produzir mudanças controladas e medir as variáveis envolvidas no fenômeno em estudo; experimentação deve também prover dados para suas futuras replicações. Entretanto, a condução e replicação de experimentos orientados a tecnologia (ou seja, experimentos cujos tratamentos são aplicados aos objetos por uma ferramenta computacional) sem suporte ferramental adequado é frequentemente uma tarefa que consome tempo e altamente sujeita a erros. Apesar de muitas técnicas terem sido propostas para auxiliar na condução de experimentos controlados, nenhuma delas trata simultaneamente (1) especificações executáveis de experimentos em alto nível de abstração; (2) execução de tratamentos e análise automatizadas a partir da especificação do experimento; e (3) garantias formais da corretude dos resultados de acordo com a especificação do experimento para experimentos orientados a tecnologia.

Objetivos: Os objetivos desse trabalho são os seguintes: (a) prover meios para especificar experimentos orientados a tecnologia em alto nível de abstração; (b) possibilitar execução e análise automatizadas dessas especificações; e (c) apresentar um modelo formal da nossa abordagem e propriedades de corretude essenciais.

Método: Nós usamos uma abordagem *Domain-Specific Modeling (DSM)* para criar uma ferramenta baseada em Web compreendendo uma *Domain-Specific Language (DSL)*, geradores de *scripts* de execução e de análise, um *framework* de suporte e uma infraestrutura de execução. Um experimentador usa a DSL para especificar um experimento usando conceitos do domínio de experimentação. A partir dessa especificação, as aplicações correspondentes aos tratamentos subjacentes são executadas, os resultados de execução são coletados e analisados e, finalmente, os resultados da análise são apresentados para o experimentador. Estabelecemos a consistência desses resultados em relação à especificação do experimento por meio da formalização e prova de propriedades de corretude essenciais da nossa ferramenta.

Resultados: Nós avaliamos empiricamente a solução em relação a automação por meio da replicação de três experimentos já publicados; avaliamos também o nível de abstração por meio de uma avaliação qualitativa. Nossa avaliação empírica mostra que a DSL é expressiva o suficiente para especificar três experimentos orientados a tecnologia selecionados e que a ferramenta de suporte pode ser usada para prover correta automação da execução e da análise a partir de especificações de experimentos orientados a tecnologia. Além disso, a DSL eleva o nível de abstração das especificações dos experimentos usando conceitos de experimentação. A prova formal de propriedades de corretude essenciais (por exemplo, corretude da geração do script de execução, otimização de recursos de execução e corretude do experimento) garante que os resultados são consistentes em relação à especificação do experimento.

Conclusão: Contribuímos com uma solução DSM e uma ferramenta correspondente compreendendo uma DSL, geradores de *scripts* de execução e de análise, um *framework* de suporte e uma infraestrutura de execução. A avaliação empírica e formal indica que a solução oferece ao experimentador abstrações e suporte de automação adequados, o que pode auxiliar na melhoria de produtividade e confiabilidade no processo de experimentação.

Palavras-chave: Experimentos controlados, Experimentos orientados a tecnologia, Modelagem específica de domínio, Linguagem específica de domínio

Abstract

Context: Experimentation is a means to produce controlled changes and to measure the variables involved in the phenomena under study; experimentation must also provide data to its further replication. However, conducting and replicating 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) runnable 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.

Objective: The objectives of this work are the following: (a) provide means to specify technology-oriented experiments at a high level of abstraction; (b) enable automated execution and automated data analysis of such specification; and (c) present a formal model of our approach and key correctness properties.

Method: We used a Domain-Specific Modeling (DSM) approach to create a Web-based tool comprising a Domain-Specific Language (DSL), execution and analysis script generators, a supporting framework, and a running infrastructure. An experimenter uses the DSL 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 of key correctness properties of our tool.

Results: We empirically evaluated the solution with respect to automation by replicating three already published experiments; we evaluated also the level of abstraction by a qualitative assessment. Our empirical evaluation shows that the DSL is expressive enough to specify three selected technology-oriented experiments and that the supporting tool can be used to enable sound automation of execution and analysis from the specification of technology-oriented experiments. In addition, the DSL raises the level of abstraction of experiment specifications by using experimentation concepts. The formal proof of key correctness properties (e.g., execution script generation soundness, execution resource optimization, and experiment soundness) assures that the results are consistent with the experiment specification.

Conclusion: We contribute a DSM approach and corresponding tool comprising a DSL, execution and analysis script generators, a supporting framework, and a running infrastructure. The empirical and formal assessment indicate that the contribution provides the experimenter with proper abstractions and automation support, which can help to improve productivity and reliability on the experimentation process.

Keywords: Controlled experiments, Technology-oriented experiments, Domain-specific modeling, Domain-specific language

List of Figures

List of Tables

List of Definitions

List of Properties

Acronyms

ANTLR ANother Tool for Language Recognition.

AST Abstract Syntax Tree.

CAV International Conference on Computer Aided Verification.

CRAN Comprehensive R Archive Network.

CTAN Comprehensive TeX Archive Network.

DSL Domain-Specific Language.

DSM Domain-Specific Modeling.

EMF Eclipse Modeling Framework.

eSEE experimental Software Engineering Environment.

FSE Joint Meeting on Foundations of Software Engineering.

Ginpex Goal-oriented INfrastructure Performance EXperiments.

HPC High Performance Computing.

IDE Integrated Development Environment.

MWE2 Modeling Workflow Engine 2.

SESE Simula Experiment Support Environment.

UI User Interface.

Contents

Chapter 1

Introduction

Empirical research is based on observations of or experimentation with real-world phenomena and its measurable changes. Experimentation is a means to produce controlled changes and to measure the variables involved in the phenomena under study. Experimentation provides further data so that other researchers can replicate original experiments and verify the results [Juristo and Moreno, 2013].

The purpose of controlled experiments is to conduct studies under strictly controlled conditions. The objective is to manipulate one or more variables and to check the effects on dependent variables. In controlled experiments, quantitative data are collected, and statistical analyses are performed [Juristo and Moreno, 2013; Wohlin et al., 2012]. Experiments can be human-oriented or technology-oriented. In the former, a person applies treatments to objects, whereas, in the latter, treatments are applied to objects by a computer-based tool [Wohlin et al., 2012].

Technology-oriented experiments are important not only in software engineering but also in other research fields, such as Bioinformatics, Engineering, Physics, and Chemistry [Chen and Chang, 2017; Houben and Lapkin, 2015; Pavlov et al., 2014; Tabatabaei, 2016]. Technology-oriented experiments can be used in several ways. First, software can be used to evaluate methodologies or approaches [Medeiros et al., 2016]. Moreover, software can be used in simulation based studies [Banks, 1999]. For example, in Engineering, systems can be simulated by using software to avoid the costs of building real systems [Tabatabaei, 2016]. In some studies, (e.g., use and occupation of the soil), evaluations cannot be performed in real-world, so they need to use simulations [Ralha et al., 2013]. 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 [Houben and Lapkin, 2015; Pavlov et al., 2014]. In this work, we focus on such technology-oriented experiments and hereafter use *experiments* to refer to this context, which includes not only *in silico* simulation based studies [Travassos and Barros, 2003] but also experiments that evaluate algorithms or computer-based tools. For example, Lanna et al. [2018] 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.

1.1 Problem Statement

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 [Juristo and Moreno, 2013; Wohlin et al., 2012]. In addition, since the scale of scientific problems has been increasing, this is reflected not only on data size but also on the complexity of the computer-based tools required to investigate such problems [Sonntag et al., 2010; Zhao et al., 2011]. Thus, such tools must run in 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 [Kephart and Chess, 2003]. Likewise, data analysis requires knowledge on statistics so that results can be correctly analyzed and interpreted. Therefore, conducting and replicating controlled experiments is an error-prone task.

First, experiments are usually specified in natural language. Since specifications in natural language are not runnable, one has to code such specification into a general purpose programming language (e.g., scripting languages). These specifications are at a low level of abstraction, though.

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 and are not runnable, whereas runnable specifications are usually written in general purpose languages, at a low-level of abstraction.

Second, to execute applications related to the treatments defined for the hypotheses of the experiment, execution scripts have to be manually written. This is often time-consuming and requires knowledge on a general purpose programming language. In addition, there may be inconsistencies between the experiment specification and its execution script. Also

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

data analysis is often complex, time-consuming, and requires knowledge on statistics so that results can be correctly analyzed and interpreted.

Problem 2

An experimenter needs to manually create the execution and the analysis scripts.

Furthermore, although there are empirical evidences, none of the existing solutions provide formal evidences of the correctness of results provided by their approaches. With correct results we mean that the results of the overall experimentation process are consistent with the experiment specification.

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.

There is a number of approaches supporting experiment conduction and replication, focusing on distinct phases of the experimentation process, and supporting human-oriented or technology-oriented experiments (Section 8.2). Although these approaches help in conducting controlled experiments, none of them simultaneously addresses runnable 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 for technology-oriented experiments. The lack of proper tool support may lead not only to extra time or resources consumption but also to incorrect results.

1.2 Proposed Solution

To address this issue, we propose a Domain-Specific Modeling (DSM) approach [Kelly and Tolvanen, 2008] supporting technology-oriented experiments. The approach is implemented as a Web-based tool and comprises a Domain-Specific Language (DSL), execution and analysis script generators, a supporting framework, and a running infrastructure. The DSL empowers researchers to specify experiments using experimentation concepts. An experimentation concept is a concept that is directly related to the experimentation domain (e.g., experimental design, treatment, experimental object, dependent variable). Execution and analysis scripts are automatically generated from the specification. Next, applications (i.e., computer-based tools) related to the treatments defined in the research hypotheses of the experiment are executed by the infrastructure, and results are then collected and analyzed by the previously generated analysis script. Finally, an analysis report is presented to the experimenter. 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. The formal proof of these properties assures the correctness of results according to the experiment specification.

We evaluated the proposed solution with respect to level of abstraction, automation, and correctness. To show that our DSL raises the level of abstraction of experiment specifications, we evaluated it by an analytical comparison between 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 our DSL raises the level of abstraction of experiment specifications. By comparing the DSL constructs with domain concepts, we found that 54.35% are high-level constructs, 15.22% are mid-level constructs, and 30.43% are low-level constructs. To show that our tool can automate execution and analysis, we replicated three already published experiments using our tool. The results suggest that the DSL is expressive enough to specify technology-oriented experiments and that the proposed tool can be used to enable sound automation of execution and analysis from the specification of technology-oriented experiments. Finally, we assured correctness by proving key formal properties of the formal specification.

1.3 Main Contributions

In summary, we make the following contributions:

- We present a Domain-Specific Modeling (DSM) approach that supports technology-oriented experiments (Chapter 4), comprising a DSL (Section 4.2), execution and analysis script generators (Sections 4.3 and 4.4), a running infrastructure (Section 4.5), and a supporting framework (Section 4.6).

- We present a Web-based tool that implements the DSM approach (Chapter 5), providing a means to specify runnable experiment specifications at a high level of abstraction; automated execution, data analysis, and results presentation.

- We empirically evaluate the practical applicability of the tool to provide automation in the experimentation process and its level of abstraction (Sections 6.1 and 6.2).

- We present a formal model of the whole procedure and proofs of correctness (Chapter 4).

1.4 Outline

The remainder of this work is organized as follows:

- Chapter 2 presents the motivation of the work and a further discussion about the problem statement.
- Chapter 3 lays conceptual foundations for the proposed research. We present concepts regarding Experimentation (Section 3.1), Domain-Specific Modeling (Section 3.2), and Running Infrastructures (Section 3.3).
- Chapter 4 presents the method, comprising a DSL (Section 4.2), execution and analysis script generators (Sections 4.3 and 4.4), a running infrastructure (Section 4.5), and a supporting framework (Section 4.6).
- Chapter 5 presents a Web-based tool that implements the DSM approach presented in Chapter 4. We present its functional view (Section 5.1), its architecture (Section 5.2), and its implementation (Section 5.3).
- Chapter 6 presents a preliminary evaluation of automation (Section 6.1) and level of abstraction (Section 6.2).
- Chapter 7 presents and analyzes the results of the evaluation of automation (Section 7.1) and level of abstraction (Section 7.2). It also presents the discussions and lessons learned (Section 7.3), as well the threats to validity (Section 7.4).
- Finally, Chapter 8 presents the conclusions, as well the limitations (Section 8.1), related work (Section 8.2), and future work (Section 8.3).

Chapter 2

Motivation

As mentioned previously, conducting an experiment is often a complex, time-consuming, and error-prone task. This complexity is inherent in all phases of the experimentation process. Validity and replicability must be addressed from the earliest phases onwards [Juristo and Moreno, 2013; Wohlin et al., 2012].

Definition and planning are critical phases. Indeed, an experiment correctly designed can gather much information from fewer executions, whereas an incorrect design can lead to extra time and resource consumption or even invalidate the results of the experiment [Juristo and Moreno, 2013; Wohlin et al., 2012]. Worse, experiments are usually specified in natural language, which may lead to ambiguity, inconsistency, and lack of information [Ciolkowski, 2012]. Specifications in natural language are not executable, therefore one has to code such specification into a general purpose programming language (e.g., a scripting language). These specifications are then at a low level of abstraction, though. In the context of technology-oriented experiments, ambiguity, inconsistency, and lack of information of natural language specifications may hamper not only the coding of low-level scripts but also future replications of the experiment.

For instance, Bak and Duggirala [2017] 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[1]. Plots were generated by Gnuplot. Listing 2.1 presents an excerpt of the corresponding execution script in Phyton.

In one of their evaluations, the authors examined the effects of optimizations for computing reachability for linear-time invariant systems with inputs. Optimizations, which correspond to treatments, are defined in Lines 11 to 20. In fact, each optimization is defined by appending distinct parameters to the tool (Lines 13 and 18). To measure

[1]http://stanleybak.com/papers/bak2017cav_repeatability.zip

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 24). The following objects are defined in Line 36 inside a loop (Line 26) until the timeout is reached (Line 33). Each treatment is applied to an experimental object in Line 32. This is executed inside a loop (Line 31), which is repeated the number of times defined in the variable `num_trials` (Line 10).

Likewise, Lanna et al. [2018] 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[2] is presented in Listing 2.2. The loop in Line 6 iterates over treatments (`strategy`) and experimental objects (`spl`). Each treatment is applied to each object (Line 11). This execution is repeated the number of times defined in `number_of_runs` (Line 25).

Although this approach to conducting an experiment works in both of the aforementioned studies, there are limitations. First, the execution scripts have to be written. This is often time-consuming and requires knowledge on a general purpose programming language, so the experimenter, in this case, should also be a programmer, which is not always the case. As a result, experimentation concepts are not clearly defined, which hampers their understanding and future replications of the experiment. 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 2.1 and `number_of_runs` in Listing 2.2 were used to represent the same concept. 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 18, Listing 2.1) could be incorrectly assigned to the treatment *Hylaa* (Line 13). Finally, 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 development of similar scripts with duplicated code for distinct experiments, which could be error-prone and time-consuming.

Also data analysis is often complex, time-consuming, and requires knowledge on statistics. Statistics is used to discover and to understand relationships between variables. Significance tests are used to check whether the differences observed in collected data are statistically significant [Juristo and Moreno, 2013]. A series of parametric and non-parametric analysis methods can be used in significance testing. Although parametric

[2]https://github.com/SPLMC/reana-evaluator/blob/master/runner.py

Listing 2.1: Excerpt of an execution script in Python [Bak and Duggirala, 2017]

```python
def main():
    '''main function'''
    measure()
    plot("opt_comparison.gnuplot")
    plot("tool_comparison.gnuplot")

def measure():
    '''run the measurements'''
    timeout_secs = 15
    num_trials = 10
    tools.append('hylaa')
    labels.append('Hylaa')
    tool_params.append('-settings settings.print_output=False')
    input_xml.append('io.xml')

    tools.append('hylaa')
    labels.append('Warm')
    tool_params.append('-settings settings.print_output=False ' +
                       'settings.opt_decompose_lp=False')
    input_xml.append('io.xml')
    for i in xrange(len(tools)):
        tool = tools[i]
        with open('out/result_{}.dat'.format(label), 'w') as f:
            step_size = 0.2

            while True:
                timeout = False
                total_secs = 0.0
                measured_secs = []

                for _ in xrange(num_trials):
                    res = e.run(print_stdout=True, run_tool=True)
                if avg_runtime > timeout_secs:
                    break

                step_size /= 1.3
```

Listing 2.2: Excerpt of an execution script in Python [Lanna et al., 2018]

```python
def run_all_analyses(number_of_runs,in_results):
    '''
    Runs all analyses for all SPLs and returns an AllStats object.
    '''
    all_stats = []
    for (spl, strategy), command_line in CONFIGURATIONS.iteritems():
        try:
            name = strategy + " ("+spl+")"
            print name
            print "----------"
            stats = run_analysis(spl, strategy, command_line,
                number_of_runs)

            all_stats.append(stats)
            print "Flushing data to replay"
            replay.save(AllStats(all_stats), in_results)
            test_hypotheses(AllStats(all_stats))
            print "===================================="
        except:
            print "Unexpected error:",
                sys.exc_info()[0],sys.exc_info()[1]
            traceback.print_tb(sys.exc_info()[2], limit=None, file=None)
            print "Error running analysis"

    return AllStats(all_stats)
def run_analysis(spl, strategy, command_line, number_of_runs):
    data = [_run_for_stats(command_line) for i in
        xrange(number_of_runs)]
    return CummulativeStats(spl, strategy, data)
```

Listing 2.3: Excerpt of a Gnuplot configuration file [Bak and Duggirala, 2017]

```
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, \
```

models can be more useful, they make stronger assumptions. Collected data must be consistent with the assumptions made by the method. Misusing an analysis method can lead to wrong results and, thus, affect the validity of the experiment [Rosenberg, 2008; Singer et al., 2008].

To make this point clearer, a Gnuplot configuration file is presented in Listing 2.3. This file 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. In addition, there may be inconsistencies between the execution script presented in Listing 2.1 and the Gnuplot file presented in Listing 2.3. For instance, each Line from 2 to 6 relates a title to the corresponding execution results. However, a title could be misassigned to a result file, what would lead to a wrong interpretation and thus incorrect results.

Another example is presented in Listing 2.4. Lanna et al. [2018] used this script[3] not only to generate plots but also to perform statistical analysis. To create this script, some knowledge on statistics was necessary. Its purpose is to check whether two data samples are significantly different. To make this comparison, either a non-parametric 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 30 would lead to the use of a parametric test when it should not be used, and, thus, invalidate the results.

Listing 2.4: Excerpt of a Python analysis script [Lanna et al., 2018]

```
1 def _compare_samples(sample1, sample2):
2     '''
3     Returns -1 if sample2 is higher, +1 if sample1 is higher or 0 if
          they are
4     not significantly different.
```

[3]https://github.com/SPLMC/reana-evaluator/blob/master/dataanalyzer.py

```
    '''
    mean1 = mean(sample1)
    mean2 = mean(sample2)
    gain = max(mean1, mean2)/min(mean1, mean2)

    if not _is_normally_distributed(sample1) or not
        _is_normally_distributed(sample2):
        normality = "Not all are normal"
        are_equal, details = _non_normal_are_equal(sample1, sample2)
    else:
        normality = "All are normal"
        are_equal, details = _normal_are_equal(sample1, sample2)

    if not are_equal:
        result = mean1 - mean2
    else:
        result = 0
    aggregated_details = (normality,
                          details,
                          {"mean 1": mean1,
                           "mean 2": mean2,
                           "gain": str(gain) + "x"})

    return result, aggregated_details
def _is_normally_distributed(sample):
    w, p = normaltest(sample)
    return p ≥ SIGNIFICANCE


def _non_normal_are_equal(sample1, sample2):
    u, p = mannwhitneyu(sample1,
                        sample2,
                        use_continuity=False)
    return p ≥ SIGNIFICANCE, ("Mann-Whitney", {"U": u, "p-value": p})


def _normal_are_equal(sample1, sample2):
    equal_vars = _variances_are_equal(sample1, sample2)
    are_equal, details = _test_normal_equality(sample1, sample2,
        equal_vars)
    return are_equal, details

def _variances_are_equal(sample1, sample2):
    stat, p = bartlett(sample1, sample2)
    return p ≥ SIGNIFICANCE
```

```

def _test_normal_equality(sample1, sample2, equal_variances):
    stat, p = ttest_ind(sample1, sample2, equal_var=equal_variances)
    method = "T-test" if equal_variances else "Welch"
    return p ≥ SIGNIFICANCE, (method, {"statistic": stat, "p-value": p})
```

Conducting experiments without proper tool support to automatically generate execution and analysis scripts is often a highly time-consuming and error-prone task. Although some techniques provide support in conducting controlled experiments (Section 8.2), none of them simultaneously provide from a specification at a high level of abstraction automated generation of execution and analysis scripts for technology-oriented experiments.

Finally, none of the existing solutions to support technology-oriented experiments provide formal evidence of the correctness of results they produce. 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. For instance, execution results could be misassigned to the underlying treatments of the hypotheses, or analysis could misplace the execution results in the analysis test, or even use an unsuitable analysis function. 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.

In the context of technology-oriented experiments, there is a lack of tool support simultaneously addressing (1) runnable 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.

Chapter 3

Background

To better understand the problem and the proposed solution, it is useful to bear in mind concepts regarding Experimentation (Section 3.1), Domain-Specific Modeling (Section 3.2), and Running Infrastructures (Section 3.3). In what follows, we lay these conceptual foundations for the proposed research.

3.1 Experimentation

Scientific research is a process of directed learning that follows an inductive-deductive process. In the inductive step, from observations of the reality, an initial model, theory or hypothesis is created. This model is then, in the deductive step, checked against the reality through observation or experimentation. When the collected data and the model fail to agree, the discrepancy can lead, by induction, to modifications in the model, and another iteration can be initiated. Experimentation is a means to produce controlled changes and to measure the variables involved in the phenomena under study. Experimentation also produces data so that other researchers can replicate the experiment and verify the results [Box et al., 2005; Juristo and Moreno, 2013].

Empirical studies can be classified as qualitative or quantitative. Qualitative studies, or exploratory studies, aim to study objects in their natural setting and let the findings emerge from the observations. In contrast, quantitative studies, or explanatory studies, aim to get a numerical relationship between several variables or alternatives under examination. In addition, quantitative studies promote comparisons and statistical analysis [Juristo and Moreno, 2013; Wohlin et al., 2012].

Empirical studies include surveys, controlled experiments, and case studies. Surveys are used to identify the characteristics of a broad population. Surveys can be conducted by using questionnaires, structured interviews, or data logging. However, surveys provide no control of the execution or the measurement. The purpose of controlled experiments is to conduct studies under strictly controlled conditions. The objective is to manipulate one or more variables and check the effects on dependent variables. In controlled experiments, quantitative data are collected, and statistical analyses are performed. In circumstances where the subjects cannot be randomly assigned to the treatments, quasi-experiments can be conducted. Case studies are done by observation of an ongoing project or activity in its real-life context in situations where the context is expected to play a central role or where the effects are expected to take a long time to appear. In these cases, controlled experiments would be inappropriate. However, in study cases, data collection and analysis are more open to interpretation and researcher bias [Easterbrook et al., 2008; Juristo and Moreno, 2013; Wohlin et al., 2012].

Travassos and Barros [2003] presented a four-staged taxonomy to classify empirical studies in software engineering:

- *In vivo*: such experiments involve people in their own environments. In Software Engineering, *in vivo* studies are executed in software development organizations throughout the development process and under real conditions and under real circumstances;
- *In vitro*: such experiments are executed in a controlled environment, such as a laboratory or a controlled community. Most *in vitro* studies are executed in universities, research centers or among selected groups of software development organizations;
- *In virtuo*: such experiments involve the interaction among participants and a computerized model of reality. The behavior of the environment with which subjects interact is described as a model and represented by a computer program. In Software Engineering, these studies are usually executed in universities and research laboratories characterized by small groups of subjects manipulating simulators;
- *In silico*: such experiments are characterized for both the subjects and real world being described as computer models. The environment is fully composed by numeric models with no human interaction.

In this work, we focus on technology-oriented experiments, which includes not only *in silico* simulation based studies but also other experiments that evaluate algorithms or computer-based tools.

3.1.1 Experimentation Concepts

We may use an experiment to evaluate our beliefs, i.e., to test a theory or hypothesis. The starting point is that we have an idea of a cause and effect relationship, which we are able to state formally in a hypothesis. Experiments are launched when we want control over the situation and want to manipulate behavior directly, precisely and systematically to compare the outcomes. Experiments may be human-oriented or technology-oriented. In human-oriented experiments, humans apply different treatments to objects, whereas in technology-oriented experiments, typically different tools are applied to different objects [Wohlin et al., 2012].

To better understand the experimentation process, it is useful to bear in mind some experimentation concepts. Some of them are also despicted in Figure 3.1.

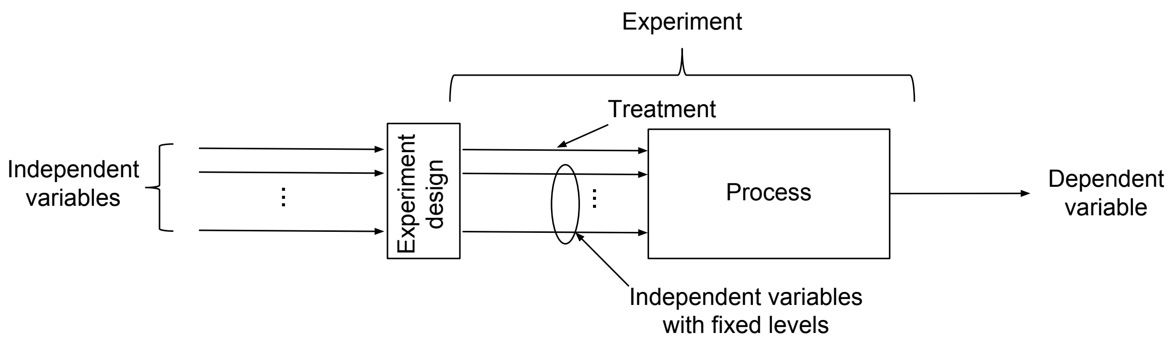


Figure 3.1: Illustration of an experiment (adapted from Wohlin et al. [2012])

Experimental subject. In human-based experiments, is the person who applies the methods or techniques to experimental objects. In contrast, in technology-based experiments, experimental subject is usually the software that applies treatments to experimental objects [Juristo and Moreno, 2013; Wohlin et al., 2012].

Experimental Object Or experimental unit. The objects on which the experiment is run [Juristo and Moreno, 2013; Wohlin et al., 2012].

Example. We want to study the effect of a new development method on the productivity of the personnel. We may have chosen to introduce an object-oriented design method instead of a function-oriented approach. The objects are the programs to be developed and the subjects are the personnel [Wohlin et al., 2012].

Independent Variable. All variables in a process that are manipulated and controlled are called independent variables [Wohlin et al., 2012].

Dependent Variable. Or response variable, is the outcome of an experiment that we want to study to see the effect of the changes in some input [Juristo and Moreno, 2013; Wohlin et al., 2012].

Example. The *dependent variable* in the previous example experiment is the productivity. *Independent variables* may be the development method, the experience of the personnel, tool support, and the environment [Wohlin et al., 2012].

Factor. The factors of an experiment are any characteristics that are intentionally varied during experimentation to examine their influence on the dependent variable [Juristo and Moreno, 2013; Wohlin et al., 2012].

Treatment. Or alternative, or level. A treatment is one particular value of a factor. This means that each treatment of a factor is an alternative for that factor [Juristo and Moreno, 2013; Wohlin et al., 2012].

Example. The factor for the example experiment above, is the development method since we want to study the effect of changing the method. We use two treatments of the factor: the old and the new development method [Wohlin et al., 2012].

Parameter. The independent variables that are controlled at a fixed level during the experiment to set a controlled environment. These are characteristics that we do not want to investigate but may influence the result of the experiment [Juristo and Moreno, 2013; Wohlin et al., 2012].

Elementary experiment. Or tests, or trials. A combination of treatment, subject and object [Juristo and Moreno, 2013; Wohlin et al., 2012].

Example. A test can be that person N (*subject*) uses the new development method (*treatment*) for developing program A (*object*). [Wohlin et al., 2012]

Experiment Design. A design of an experiment describes how the tests are organized and run. The designs range from simple experiments with a single factor to more complex experiments with many factors or treatments [Wohlin et al., 2012].

3.1.2 Experimentation Process

Conducting an experiment is a complex and time-consuming task. The experimenter has to prepare, conduct, and analyze experiments properly. A process provides support in setting up and conducting an experiment [Juristo and Moreno, 2013; Wohlin et al., 2012]. In what follows, we describe an experimentation process (Figure 3.2). Although this process is described for controlled experiments, it could be adapted to other empirical strategies. The phases of the process are: scoping (definition); planning; operation (execution); analysis and interpretation; and presentation and package.

In scoping, or definition phase, the experimenter describes the goals of the experiment, i.e., what the experiment aims to investigate and its motivation. The goal is formulated from the problem to be solved. Then, a hypothesis is clearly defined in terms of what variables are going to be examined [Juristo and Moreno, 2013; Wohlin et al., 2012].

In planning phase, the hypothesis is formally stated, including a null hypothesis and an alternative hypothesis. The context is thoroughly defined and variables and scales of measure are determined. Next, the experimental design is defined. This design defines an experiment as a set of tests. Each test is a combination of treatment, subject, and object.

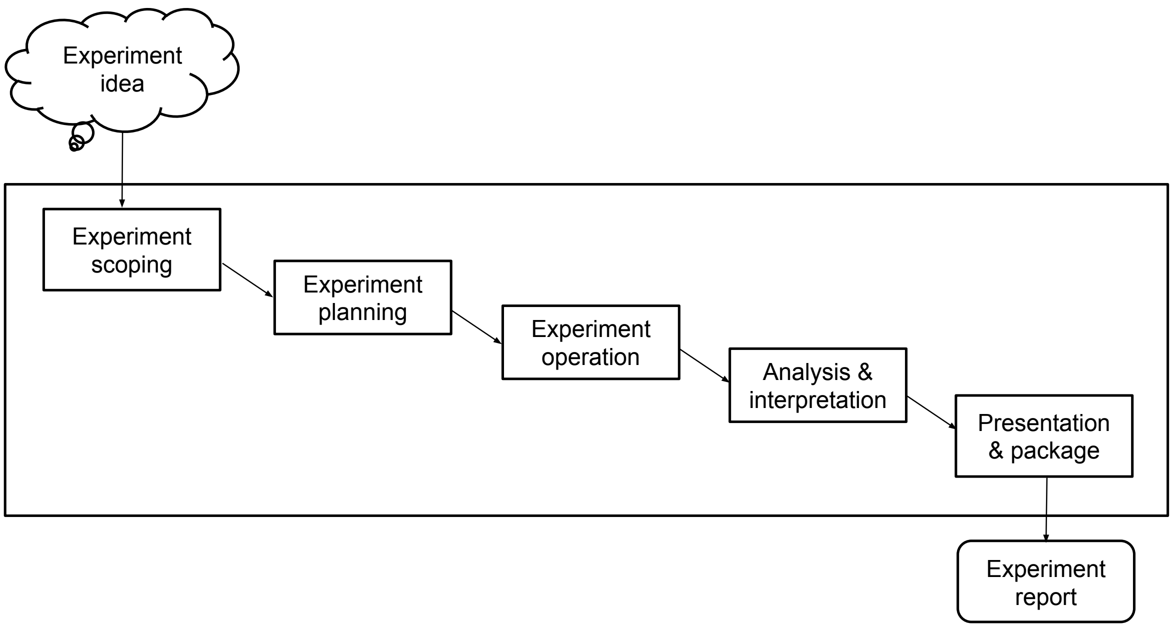


Figure 3.2: Experiment Process (adapted from Wohlin et al. [2012])

In addition, design defines how many tests are going to be run [Juristo and Moreno, 2013; Wohlin et al., 2012].

Design and analysis are closely related since the statistical analyses that are going to be applied depend on the chosen design and measurement scales defined in design. In the same way, to design an experiment, the experimenter has to take into account which statistical analysis must be performed to reject the null hypothesis [Juristo and Moreno, 2013; Wohlin et al., 2012].

The plan must consider validity of the results. Easterbrook et al. [2008] classifies validity as construct validity, internal validity, external validity, and reliability. Construct validity focuses on whether the theoretical constructions are correctly interpreted and measured. Internal validity focuses on the design and whether the results actually follow the data. External validity is related to the generalization of the results. Finally, reliability is related to bias in the research. Independent researchers repeating the experiment must yield the same results.

Planning phase is crucial to provide further replications of the experiment. In addition, there are several types of replication. Despite the use of the term *replication* with distinct meanings in literature, replications usually fall into three groups: replications that vary little or not at all with respect to the reference experiment; replications that do vary but still follow the same method as the reference experiment; and replications that use different methods to verify the reference experiment results [Gómez et al., 2010]. In this work, we use the term replication with the meaning of exact replication: "An exact replication is one in which the procedures of an experiment are followed as closely as possible to determine whether the same results can be obtained" [Shull et al., 2008].

During operation, or execution phase, the first step is to prepare the subjects, the instruments, and the material needed. Then, the subjects apply the treatments to the objects according to the experimental design, and data are collected. Finally, the collected data must be validated to make sure that they are correct and provide a valid picture of the experiment [Juristo and Moreno, 2013; Wohlin et al., 2012].

In analysis and interpretation phase, data collected in execution are analyzed and interpreted. First, descriptive statistics is applied to provide a visualization of the data and help the experimenter to understand and interpret the data informally. Next, data reduction is performed, which consists in considering whether the data set should be reduced, either by removing data points or by reducing the number of variables. After that, based on the design and on the nature of the data, hypothesis tests are performed. Finally, results are interpreted [Juristo and Moreno, 2013; Wohlin et al., 2012].

Presentation and package is the documentation and presentation of the results. It can be done in a paper, in a technical report, or in a package for replication of the experiment, depending on the purpose of the experiment. A common problem is that experiments are poorly or heterogeneously reported [Jedlitschka et al., 2008; Juristo and Moreno, 2013], which hampers further replications, due to difficulty to find context information from the original experiment [Wohlin et al., 2012].

3.1.3 Data Analysis and Presentation Tools

As mentioned before, data analysis is complex, time-consuming, and requires knowledge on Statistics so that results can be correctly analyzed and interpreted. Statistics is used to discover and to understand relationships between variables. Significance tests are used to check whether the differences observed in collected data are statistically significant [Juristo and Moreno, 2013]. Misusing an analysis method can lead to wrong results and, thus, affect the validity of the experiment [Rosenberg, 2008; Singer et al., 2008]. Equally important is presentation and package. Not only must data analysis be correct but also all scripts and raw data must be presented, so that other researchers can easily re-analyze the data either using the same data analysis technique to verify that no errors were made during the data analysis phase, or with other analysis techniques to verify whether similar findings can be obtained using the same data of a previous experiment [Gómez et al., 2010; Madeyski and Kitchenham, 2017]. For this reason, using proper tool supporting data analysis and presentation of results is essential to achieve valid and reproducible results.

In Computational Science there is an initiative called reproducible research. Reproducible research refers to the idea that the ultimate product of research is the paper plus the entire environment used to produce the results in the paper (data, software, etc.). Reproducible research aims that anything in a scientific paper should be reproducible by the reader, including results, plots and graphs [Kovacevic, 2007; Madeyski and Kitchenham, 2017].

Furthermore, Madeyski and Kitchenham [2017] discussed the use of Reproducible Research to address some problems found in empirical software engineering research, particularly issues related to validity and reproduction of data analysis. The authors suggested the use of a set of free and open-source tools to use in practice to produce reproducible research, including R [Team, 2018], LaTeX [Lamport, 1994], and Sweave [Leisch, 2002].

R is a programming language and free open-source (GNU-licensed) environment derived from the earlier S language developed at Bell Labs. Its main application fields are statistical computing and graphics. The main repository for software products developed by the R community, known as packages, is the Comprehensive R Archive Network (CRAN)[1]. In addition, a large number of packages also exists independently on code repositories like GitHub [Plakidas et al., 2016]. R is important for Reproducible Research because the use of a statistical language and open source environment provides more traceability to the details of the statistical analysis than a closed source statistical package. In addition, typing an R script is more reproducible and easier to communicate than using the point-and-click user interface often adopted in other statistical packages [Madeyski and Kitchenham, 2017].

LaTeX is an ideal language for representing mathematical and statistical equations [Madeyski and Kitchenham, 2017]. LaTeX is an extremely popular system for typesetting documents in the scientific and academic communities, and it is extensively used in industry. An experienced user can define commands to represent mathematical structures, for instance, and use these commands to keep the format of the mathematical structures consistent along all the document. In addition, an user do not have to worry about formatting while writing a document. Formatting decisions can be made and changed at any time [Lamport, 1994]. Besides the default functions provided by LaTex, the LaTeX community also creates and makes available additional packages in Comprehensive TEX Archive Network (CTAN)[2].

Sweave embeds the statistical analysis in LaTex. The purpose is to create dynamic reports, which can be updated automatically if data or analysis change, while using standard tools for both data analysis and word processing. Sweave is written in the S language but either the open source R or the commercial Splus[3] can be used for statistical

[1]https://cran.r-project.org/
[2]https://ctan.org/
[3]http://www.insightful.com

data analysis. Sweave is part of every R installation (version 1.5.0 or higher). Sweave files are easy to write and offer the full power of LaTex for high-quality typesetting. Storing code and documentation in a single source file makes research completely reproducible since all results can easily be verified and regenerated [Leisch, 2002].

The use of the aforementioned tools supports the iterative nature of research where results are often revised and re-analyzed by providing mechanisms to easily update tables and figures, thus keeping reports updated if data are changed or new analyses are required [Madeyski and Kitchenham, 2017].

3.2 Domain-Specific Modeling

Models are used in Software Engineering to raise the level of abstraction and hide implementation details. DSM aims to raise the level of abstraction by using domain concepts. The solution is specified by using a DSL, which is a language optimized to a given class of problems related to the domain. From the language, complete code is then generated. This is possible because both language and generator are domain specific [Kelly and Tolvanen, 2008; Voelter et al., 2013].

Domains can be horizontal or vertical. Horizontal domains are technical domains, such as persistence, user interface, communication, or transactions. In contrast, vertical domains are business domains, such as telecommunication, banking, robot control, insurance, or retail. Each DSM solution focuses on a narrow domain because it offers substantial gains in expressiveness with corresponding gains in productivity and reduced maintenance costs [Kelly and Tolvanen, 2008; Mernik et al., 2005].

Using DSM brings many benefits. First, due to the higher level of abstraction and to code generation, productivity is improved. Less code must be written and read, and common code is reused. Improved productivity also leads to shorter time-to-market and lower development costs. Second, the product quality is improved. Modeling languages can include correctness rules of the domain, which makes finding and correcting bugs easier and cheaper. Finally, DSM hides the solution complexity and implementation details. Since the technical aspects of the solution are designed and implemented during the DSM construction, domain experts can use the DSL to provide specifics solutions at a high level of abstraction [Fowler, 2010; Kelly and Tolvanen, 2008; Voelter et al., 2013].

3.2.1 Domain Specific Modeling Architecture

To get the benefits from using DSM, Kelly and Tolvanen [2008] proposed a three-layer architecture composed by DSL, Generator, and Framework.

A DSL is a small, usually declarative, language that offers expressive power and an abstraction mechanism to deal with complexity in a given domain. Instead of using concepts of a programming language, the solution is specified by using domain concepts. Using DSL improves communication between developers and domain experts. In addition, DSL allows domain experts to actively take part in the development process [Kelly and Tolvanen, 2008; van Deursen et al., 2000].

The code generator plays a central role in DSM. A generator specifies how information is extracted from models and transformed into code. Instead of having source code, developers have source model, which are specified by using the DSM. Generated code must be complete code. As a result, no generated code should be touched by hand. Any changes must be done either in the models or in the generator. In addition, the generator target is not limited to programming language code. From the model, the generator can also generate another model or a text file in any format [Fowler, 2010; Kelly and Tolvanen, 2008].

Although generated code is complete in the sense that no further modification is required, generated code, due to its narrow focus, usually does not provide a whole solution for the problem. Some extra utility code or components may be necessary. These components can be new or existing code. The domain framework provides the interface between generated code and the underlying platform [Kelly and Tolvanen, 2008].

3.2.2 DSL Implementation

Implementing a DSL means developing a program that is able to read text written in that DSL, parse it, process it, and then possibly interpret it or generate code in another language. Xtext is an Eclipse framework for implementing programming languages and DSLs. It covers all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a complete Eclipse Integrated Development Environment (IDE) integration with all the typical IDE features, such as syntax highlighting, background validation, error markers, content assist, quick-fixes, and automatic build [Bettini, 2016].

To start a DSL implementation, Xtext needs only a grammar specification similar to ANother Tool for Language Recognition (ANTLR). From this specification, Xtext automatically generates the lexer, the parser, the Abstract Syntax Tree (AST) model, the construction of the AST to represent the parsed program, and the Eclipse editor with all the IDE features. AST is a convenient representation in memory of a program and represents the abstract syntactic structure of the program [Bettini, 2016].

Xtext uses the Modeling Workflow Engine 2 (MWE2) DSL to configure the generation of its artifacts. During the MWE2 workflow execution, Xtext generates artifacts related to the User Interface (UI) editor for the DSL, and derive an ANTLR specification from the

Xtext grammar with all the actions to create the AST while parsing. Xtext automatically infers the Eclipse Modeling Framework (EMF) meta-model for the language, and using this meta-model, generates the classes for the nodes of the AST [Bettini, 2016].

The EMF is a modeling framework that provides code generation facilities for building tools and applications based on structured data models. Most of the Eclipse projects that in some way deal with modeling are based on EMF since it simplifies the development of complex software applications with its mechanisms [Steinberg et al., 2008].

Parsing a program is only the first stage in a programming language implementation. In particular, the overall correctness of a program cannot always be determined during parsing. Some additional static analysis can be performed only when other program parts are already parsed. Actually, the best practice is to do as little as possible in the grammar and as much as possible in validation. In Xtext, these validations are implemented using a validator that performs constraint checks on the elements of an EMF model [Bettini, 2016].

After a program written in the DSL has been parsed and validated, typically code in another language, for example, Java code, a configuration file, XML, or a text file, is generated from the parsed EMF model, that is, the AST of that program. In all of these cases, one needs to write a code generator. Then, Xtext automatically integrates the code generator into the Eclipse build infrastructure [Bettini, 2016].

Although the validators and code generators can be implemented in Java, Xtend provides useful mechanisms for writing code generators, for example, multi-line template expressions, in addition to powerful features that make model visiting and traversing really easy, straightforward, and natural to read and maintain. Xtend is a statically typed language that uses the Java type system, including Java generics and Java annotations. Xtend programs are translated into Java, and Xtend code can access all the Java libraries; thus Xtend and Java can cooperate seamlessly [Bettini, 2016].

For instance, using Xtext, Freire et al. [2013] created a DSL named ExpDSL to specify controlled experiments in software engineering. ExpDSL comprises four views: process view, which allows defining the procedures of data collection from the experiment participants; metric view, used to define the metrics that have to be collected during the experiment execution; experimental plan view, used to define the experimental plan; and questionnaire view, which allows defining questionnaires in order to collect quantitative and qualitative data from participants of the experiment.

Their approach also comprises 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 and the researchers running the experiment can monitor the activities performed by the participants. The purpose of the workflow is to guide participants in human-based experiments by providing instructions for their tasks.

Complementary to Xtext, DSLFORGE [Lajmi et al., 2014] is a framework for the generation of web textual DSL editors from the DSL grammar, validators, and code generators created using Xtext. The generated editors are packaged into workbench web applications based on Eclipse Remote Application Platform (RAP) and let users create, edit, and launch transformations from models. These on-line editors are also easily customizable and extensible.

3.3 Running Infrastructures

Since the scale of scientific problems has been increasing, this is reflected not only on data size but also on the complexity of the computer-based tools required to investigate such problems [Sonntag et al., 2010; Zhao et al., 2011]. Thus, such tools must run in an infrastructure that provides computing power, data storage, and network resources. Among many others, Docker Containers (Section 3.3.1) and Autonomic Computing (Section 3.3.2) help in providing an infrastructure to run technology-oriented experiments.

3.3.1 Docker Containers

When it comes to technology-oriented experiments, the concept of Reproducible Research discussed in Section 3.1.3 must also consider the execution environment. Crucial scientific processes, such as replicating the results, extending the approach or testing the conclusions in other contexts, or even merely installing the software used by the original researchers can become immensely time-consuming if not impossible [Boettiger, 2015].

Docker is a platform supporting container for developing, shipping and running distributed applications. Although Docker has largely focused on the needs of businesses in deploying web applications and the potential for a lightweight alternative to full virtualization, these features have potentially important implications for systems research in the area of scientific reproducibility [Boettiger, 2015; Chung et al., 2016].

A Docker based approach works similarly to a virtual machine image in addressing the "Dependency Hell" problem by providing other researchers with a binary image in which all the software has already been installed, configured and tested. A key difference between Docker images and virtual machines is that the Docker images share the Linux kernel with the host machine. Sharing the Linux kernel makes Docker more light-weight and higher performing than complete virtual machines. On the other hand, any Docker image must be based on a Linux system with Linux-compatible software, which includes R, Python, Matlab, and most other scientific programming needs. When running a job, each container is assigned a unique PID; it can be observed equivalently as a process at the view of host machine [Boettiger, 2015; Chung et al., 2016].

The steps necessary to build up a Docker image are documented within a Dockerfile. While machine images can be very large, a Dockerfile is just a simple script file that can be easily stored and shared. Dockefile is also suited for use with a version management system such as Subversion or Git. In addition, it is straightforward for other users to extend or customize the resulting image by editing the script directly [Boettiger, 2015].

3.3.2 Autonomic Computing

The term Autonomic Computing was first used by IBM in 2001 to describe self-managed systems, in an analogy with the human autonomic nervous system. Autonomic System is also refereed as Self-Managing System or Self-Adaptive System [Huebscher and McCann, 2008].

The growing complexity of information technology infrastructures threatens the benefits the systems aims to provide. Installing, configuring, and executing applications in such infrastructures are complex, time-consuming, and error-prone tasks even for experts. Autonomic computing aims to decrease human involvement in managing resources [Horn, 2001; Huebscher and McCann, 2008; Kephart and Chess, 2003]

Autonomic Computing hides the complexity of managing infrastructure resources from users since they have only to specify the goals of the infrastructure instead of specifying how to achieve them. From the goals and the high level policies, autonomic systems must be capable of running themselves, adjusting to varying circumstances and anticipating resource needs [Horn, 2001].

The essence of Autonomic Computing is self-management. This involves self-configuration, self-optimization, self-healing, and self-protection [Horn, 2001; Kephart and Chess, 2003]. First, with self-configuration, autonomic systems will configure themselves automatically according to the user's goals and high-level policies. This includes installing, configuring, and integrating large complex systems. Second, self-optimization means that autonomic system will continuously seek for improving their operation. Furthermore, an autonomic system with self-healing will detect, diagnose, and repair localized problems resulting from failures in hardware or software. Finally, self-protection means that an autonomic system will protect the whole system against large-scale problems arising from malicious attacks or cascade failures. In addition, the system will, based on early reports from sensors, anticipate problems to avoid or mitigate them.

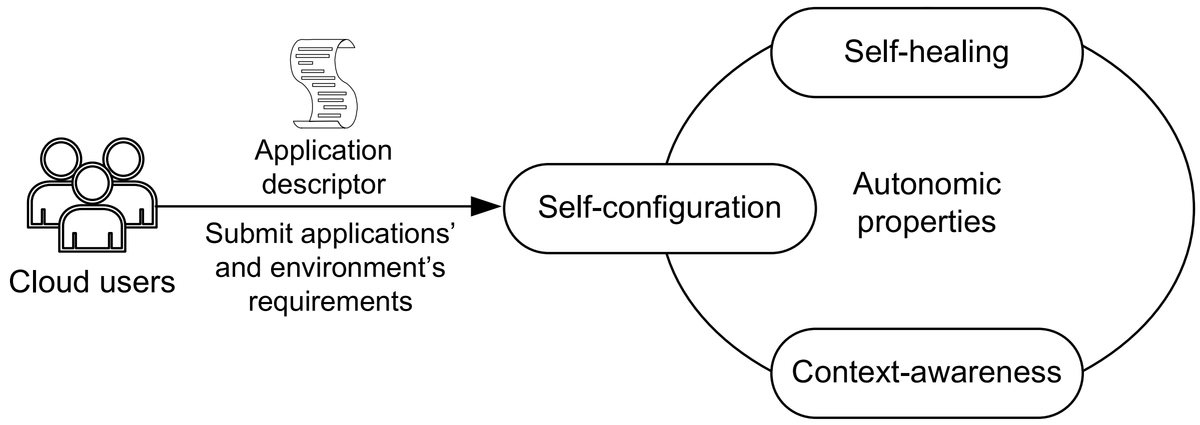


Figure 3.3: Autonomic properties implemented by Dohko [Leite et al., 2016]

For instance, Leite et al. [2017] proposed an autonomic and goal-oriented system, namely Dohko. As depicted in Figure 3.3, Dohko follows a declarative strategy to provide self-configuration, self-healing, and context-awareness. The users describe their applications, requirements, and constraints in an Application Descriptor. Then, from the *Application Descriptor*, the system creates and configures the whole computing environment, monitors the availability and state of the nodes through self-healing, and connects the nodes taking into account their locations. This allows users to concentrate on their objectives rather than on dealing with cloud or system administration issues.

In this chapter, we layed the conceptual foundations for our research, including Experimentation (Section 3.1), Domain-Specific Modeling (Section 3.2), and Running Infrastructures (Section 3.3). These concepts are useful to better understand the proposed solution, which is described in the following chapters.

Chapter 4

Method

Our objective is to provide a solution simultaneously addressing (1) runnable 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 adress the aforementioned problems, we present a DSM-based solution, which comprises a DSL (Section 4.2), an experiment execution script generator (Section 4.3), an analysis script generator (Section 4.4), a running infrastructure (Section 4.5), and a supporting framework (Section 4.6). 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 4.3 and 4.4).

4.1 Overview

We present a DSM-based solution as depicted in Figure 4.1. Initially, we created a DSL, execution and analysis scripts generators, and a supporting framework. The DSL is then used by other researchers to specify an experiment.

In the DSL, 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.

The DSL 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.

The running infrastructure executes the experiment execution script producing a series of data points. The framework monitors execution and collects partial results. After execution, the framework automatically collects and analyzes 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 lab 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 lab package.

4.2 The DSL

Following an action research method [Easterbrook et al., 2008], we developed the DSM solution inspired by the experimental challenges reported by a colleague in our research group [Lanna et al., 2018]. Our DSL is partially based on ExpDSL [Freire et al., 2013], 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 and successfully used in a number of experiments [Freire et al., 2014].

The syntax of the DSL, containing the main constructs of the language, is presented in Listing 4.1. 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. The only exception is $\overline{EX}$ (Line 6, Listing 4.2), which actually is a set of executions EX. We also assume the existence of primitive types, such as $String$, $PosInt$, and $Float$.

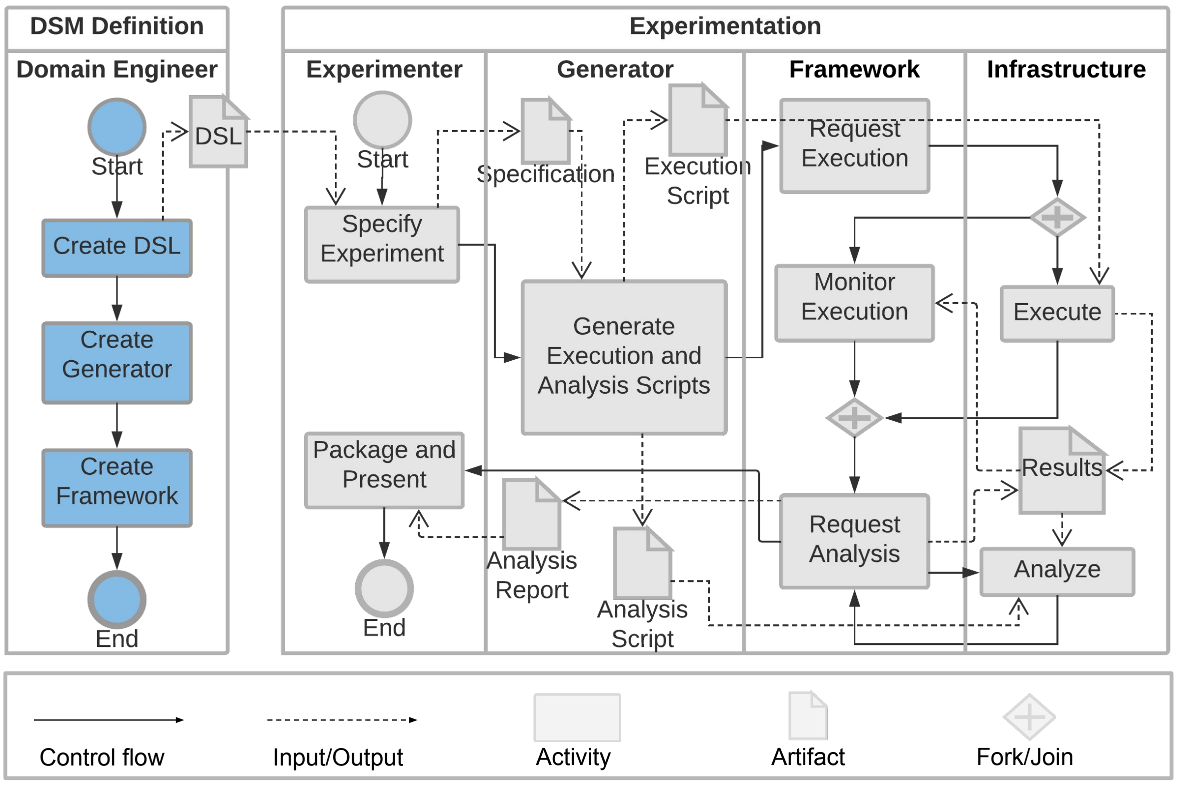


Figure 4.1: Proposed DSM-based solution

Listing 4.1: 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 $DV ::= \{name : String,\ \ instrument : String\}$

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 instrument defines how to measure the corresponding dependent variable in the infrastructure. This model description (abstract syntax) is complemented by the following definition of well-formedness:

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. 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.

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

4.3 Experiment Execution Script Generation and Experiment Execution

An execution script ES comprises a list of applications A (Line 1, Listing 4.2). 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 of its execution (Line 4).

Listing 4.2: Execution Script and Execution Model

$$\begin{array}{l}
ES ::= \{applications : \overline{A}\} \\
A ::= \{instrument : String,\ \ command : String,\ \ argument : String\} \\
EX ::= \{dependentVariable : DV,\ \ treatment : T,\ \ object : O\} \\
ER ::= \{instrument : String,\ \ command : String,\ \ argument : String,\ \ value : Float\} \\
generateExecutionScript : E \rightarrow ES \\
applyDesign : D \ \times\ \overline{H} \ \times\ \overline{O} \ \rightarrow \overline{EX} \\
generateApplication : EX \ \rightarrow A \\
execute : ES \rightarrow \overline{ER}
\end{array}$$

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 set of executions *EX*. 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).

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):

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

Algorithm 1 Execution Script Generation

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

17: function APPLYDESIGN($design, hypotheses, objects$)
18:    $executions \leftarrow$ new $Set$        ▷ We are using Set since $executions$ must not contain repetitions
19:    for all $hypothesis \in hypotheses$ do
20:        $t1 \leftarrow hypothesis.treatment_1$
21:        $t2 \leftarrow hypothesis.treatment_2$
22:        $relatedTreatmentsAndObjects \leftarrow design.designFunction(\{t_1, t_2\}, objects)$
23:        for all $pairTreatmentObject \in relatedTreatmentsAndObjects$ do
24:            $execution \leftarrow$ new $EX$
25:            $execution.treatment \leftarrow pairTreatmentObject.treatment$
26:            $execution.object \leftarrow pairTreatmentObject.object$
27:            $execution.dependentVariable \leftarrow hypothesis.dependentVariable$
28:            insert $execution$ into $executions$
29:        end for
30:    end for
31:    return $executions$
32: end function

33: function GENERATEAPPLICATION($execution$)
34:    $application \leftarrow$ new $A$
35:    $application.command \leftarrow execution.treatment.command$
36:    $application.argument \leftarrow execution.object.argument$
37:    $application.instrument \leftarrow execution.dependentVariable.instrument$
38:    return $application$
39: end function
```

script.

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

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

$\square$

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. Specifically, 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.

$$\begin{aligned}
&\forall e : E \cdot \mathit{wf}(e) \implies \forall h \in e.\mathit{hypotheses}\cdot\\
&\forall (t, o) \in e.\mathit{design}.\mathit{designFunction}(\{h.\mathit{treatment}_1, h.\mathit{treatment}_2\}, e.\mathit{objects})\cdot\\
&\exists_{=n} a \in \mathit{generateExecutionScript}(e).\mathit{applications} \mid\\
&a.\mathit{instrument} = h.\mathit{dependentVariable}.\mathit{instrument} \land\\
&a.\mathit{command} = t.\mathit{command} \land\\
&a.\mathit{argument} = o.\mathit{argument}
\end{aligned}$$

Proof sketch. 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, toghether 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).

$\square$

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. 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.

$$\begin{aligned}
&\forall e : E \cdot \mathit{wf}(e) \implies \forall a \in \mathit{generateExecutionScript}(e).\mathit{applications}\cdot\\
&\exists h \in e.\mathit{hypotheses}, t \in \{h.\mathit{treatment}_1, h.\mathit{treatment}_2\}, o \in e.\mathit{objects} \mid\\
&a.\mathit{instrument} = h.\mathit{dependentVariable}.\mathit{instrument} \wedge\\
&a.\mathit{command} = t.\mathit{command} \wedge\\
&a.\mathit{argument} = o.\mathit{argument} \wedge\\
&(t, o) \in e.\mathit{design}.\mathit{designFunction}(\{h.\mathit{treatment}_1, h.\mathit{treatment}_2\}, e.\mathit{objects})
\end{aligned}$$

Proof sketch. 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 3–10) 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 4.4).

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

Definition 3 (Infrastructure semantics)**.**

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

Algorithm 2 Experiment Execution

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

4.4 Analysis Script Generation and Analysis

An analysis script AS (Line 1, Listing 4.3) 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. 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.

Listing 4.3: Analysis Script and Analysis Model

1 $AS ::= \{hypothesesTests : \overline{HT}\}$
2 $HT ::= \{hypothesisName : String,\ analysisTests : \overline{AT}\}$
3 $AT ::= \{analysisFunction : \overline{ER} \times \overline{ER} \to AR,\ parameter_1 : P,\ parameter_2 : P\}$
4 $P ::= \{instrument : String,\ command : String,\ argument : String\}$
5 $HR ::= \{hypothesisName : String,\ testResults : \overline{TR}\}$
6 $TR ::= \{argument : String,\ analysisResult : AR\}$
7 $AR ::= \{result : String\}$
8 $generateAnalysisScript : E \to AS$

```
9  $generateHypothesisTest : D \times H \times \overline{O} \rightarrow HT$
10 $generateAnalysisTest : H \times O \rightarrow AT$
11 $generateParameter : D \times T \times O \rightarrow P$
12 $suitableFunction : H \rightarrow (\overline{ER} \times \overline{ER} \rightarrow AR)$
13 $analyze : \overline{ER} \times AS \rightarrow \overline{HR}$
14 $analyzeHypothesis : HT \rightarrow HR$
15 $applyAnalysisTest : AT \rightarrow TR$
16 $filterResults : \overline{ER} \times P \rightarrow \overline{ER}$
```

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 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 [Box et al., 2005; Juristo and Moreno, 2013]. Since this analysis function is provided uniquely based on the hypothesis, it is actually a procedure with parametric and non-parametric 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 (non-parametric) 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 dependent 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.

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.

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

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

□

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 $\overline{HR}$. Each *hypothesisTest* of the analysis script (Lines 3–6) is analyzed by the function *analyzeHypothesis* (Line 4).

Algorithm 3 Analysis Script Generation

```
function GENERATEANALYSISSCRIPT($experimentSpecification$)
    $hypothesesTests \leftarrow$ new $List$
    $design \leftarrow experimentSpecification.design$
    $hypotheses \leftarrow experimentSpecification.hypotheses$
    $objects \leftarrow experimentSpecification.objects$
    for all $hypothesis \in hypotheses$ do
        $hypothesisTests \leftarrow generateHypothesisTest(design, hypothesis, objects)$
        insert $hypothesisTests$ into $hypothesesTests$
    end for
    $analysisScript \leftarrow$ new $AS$
    $analysisScript.hypothesesTests \leftarrow hypothesesTests$
    return $analysisScript$
end function

function GENERATEHYPOTHESISTEST($design, hypothesis, objects$)
    $analysisTests \leftarrow$ new $List$
    $executions \leftarrow applyDesign(design, \{hypothesis\}, objects)$
    for all $execution \in execution$ do
        $object \leftarrow execution.object$
        $visitedObjects \leftarrow$ new $List$
        if $object \notin visitedObjects$ then          ▷ Creates only one analysis test per object
            $analysisTest \leftarrow generateAnalysisTest(hypothesis, object)$
            insert $analysisTest$ into $analysisTests$
            insert $object$ into $visitedObjects$
        end if
    end for
    $hypothesisTest \leftarrow$ new $HT$
    $hypothesisTest.hypothesisName \leftarrow hypothesis.name$
    $hypothesisTest.analysisTests \leftarrow analysisTests$
    return $hypothesisTest$
end function

function GENERATEANALYSISTEST($hypothesis, object$)
    $dv \leftarrow hypothesis.dependentVariable$
    $t_1 \leftarrow hypothesis.treatment_1$
    $t_2 \leftarrow hypothesis.treatment_2$
    $analysisTest \leftarrow$ new $AT$
    $analysisTest.analysisFunction \leftarrow suitableFunction(hypothesis)$
    $analysisTest.parameter_1 \leftarrow generateParameter(dv, t_1, object)$
    $analysisTest.parameter_2 \leftarrow generateParameter(dv, t_2, object)$
    return $analysisTest$
end function
```

```
41: function GENERATEPARAMETER($dependentVariable, treatment, object$)
42:     $parameter \leftarrow$ new $P$
43:     $parameter.instrument \leftarrow dependentVariable.instrument$
44:     $parameter.command \leftarrow treatment.command$
45:     $parameter.argument \leftarrow object.argument$
46:     return $parameter$
47: end function
```

This function (Line 9) analyzes all the *analysisTests* (Lines 11–14) of that *hypothesisTest*. Each *analysisTest* is analyzed 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 an experiment consists of the confirmation/rejection of its hypotheses.

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

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.

Algorithm 4 Analysis

```
1: function ANALYZE($executionResults$, $analysisScript$)
2:     $hypothesesResults \leftarrow$ new $List$
3:     for all $hypothesisTest \in analysisScript.hypothesesTests$ do
4:         $hypothesisResults \leftarrow analyzeHypothesis(hypothesisTest)$
5:         insert $hypothesisResults$ into $hypothesesResults$
6:     end for
7:     return $hypothesesResults$
8: end function

9: function ANALYZEHYPOTHESIS($hypothesisTest$)
10:     $testResults \leftarrow$ new $List$
11:     for all $analysisTest \in hypothesisTest.analysisTests$ do
12:         $testResult \leftarrow applyAnalysisTest(analysisTest)$
13:         insert $testResult$ into $testResults$
14:     end for
15:     $hypothesisResults \leftarrow$ new $HR$
16:     $hypothesisResults.hypothesisName \leftarrow hypothesisTest.hypothesisName$
17:     $hypothesisResults.testResults \leftarrow testResults$
18:     return $hypothesisResults$
19: end function

20: function APPLYANALYSISTEST($analysisTest$)
21:     $results_1 \leftarrow filterResults(executionResults, analysisTest.parameter_1)$
22:     $results_2 \leftarrow filterResults(executionResults, analysisTest.parameter_2)$
23:     $analysisResult \leftarrow analysisTest.analysisFunction(results_1, results_2)$
24:     $testResult \leftarrow$ new $TR$
25:     $testResult.argument \leftarrow analysisTest.parameter_1.argument$
26:     $testResult.analysisResult \leftarrow analysisResult$
27:     return $testResult$
28: end function

29: function FILTERRESULTS($results, parameter$)
30:     $filteredResults \leftarrow$ new $List$
31:     for all $result \in results$ do
32:         if $result.instrument = parameter.instrument \wedge result.command = parameter.command \wedge result.argument = parameter.argument$ then
33:             insert $result$ into $filteredResults$
34:         end if
35:     end for
36:     return $filteredResults$
37: end function
```

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. In addition, execution data are produced by executing a sound execution script generated from the experiment specification. For each hypothesis, the analysis function is suitable to analyze it, and each parameter of the analysis function corresponds to a set of data resulting from applying each treatment to an object, according to the experimental design, and measured by the corresponding instrument. The parameters are provided to the analysis function in the correct order. Moreover, execution data are produced by executing a sound execution script generated from a well-formed experiment specification.

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

$\mathit{HR} \leftrightarrow H$ is a bijection between hypotheses results HR and hypotheses H. Given a hypothesis $h : H$, $hr : \mathit{HR}$ is its corresponding result.

Likewise, $\mathit{TR} \leftrightarrow \mathit{hObjects}$ is bijection between test results TR and the objects resulting of applying the design function to the treatments of a given hypothesis and the objects. We also use a helper function $\mathit{objects} : \overline{(T, O)} \rightarrow \overline{O}$.

Proof sketch. 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) and *at.parameter*$_2$ (Line 22); *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 treatment of the same hypothesis (Lines 37 and 38, Algorithm 3).

Each hypothesis test of the analysis script as is generated from a hypothesis defined in e (Line 7, Algorithm 3). This establishes a bijection between $e.hypotheses$ and $as.hypothesesTests$:

$$H \leftrightarrow HT \; generateHypothesisTest(H, ...) : HT$$

The analysis of each $as.hypothesesTests$ (Line 4, Algorithm 4) results in a hypothesis result HR. This establishes a bijection between HT and HR:

$$HT \leftrightarrow HR \; analyzeHypothesis(HT) : HR$$

So, by transitivity, or composition of functions, there is also a bijection between H and HR:

$$H \leftrightarrow HR \; analyzeHypothesis(generateHypothesisTest(H, ...)) : HR$$

For each object related to a hypothesis by the design function (Line 16, Algorithm 3), an analysis test is created (Line 21). This establishes a bijection between the related objects ($hObjects$) and the analysis tests (AT):

$$hObjects \leftrightarrow AT \; generateAnalysisTest(O, ...) : AT$$

Each analysis test is analyzed (Line 12, Algorithm 4), resulting in a test result TR. This establishes a bijection between AT and TR:

$$AT \leftrightarrow TR \; applyAnalysisTest(AT) : TR$$

So, by transitivity, or composition of functions, there is also a bijection between $hObjects$ and TR:

$$hObjects \leftrightarrow TR \; applyAnalysisTest(generateAnalysisTest(O, ...)) : TR$$

□

4.5 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 send 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. In addition, the running infrastructure must be able to run an analysis script and present the corresponding analysis report.

4.6 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 4.2 shows how the supporting framework interacts with the other elements of our DSM solution. By using 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 using 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 using 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.

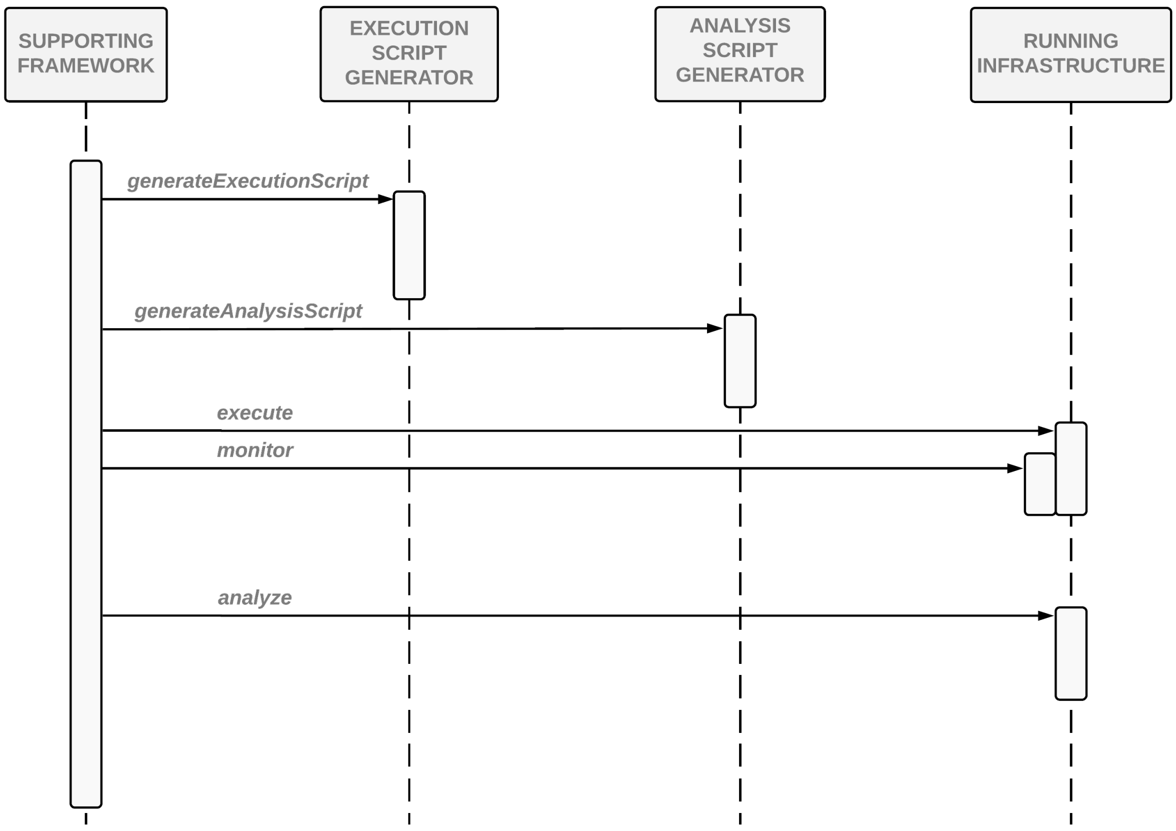


Figure 4.2: Supporting Framework Interactions

Chapter 5

Tool Support

In this section, we present a Web-based tool that implements the DSM approach (Chapter 4), providing a means to specify runnable specifications at a high level of abstraction; automated execution, data analysis, and results presentation. We present its functional view (Section 5.1), its architecture (Section 5.2), and its implementation (Section 5.3).

5.1 Functional View

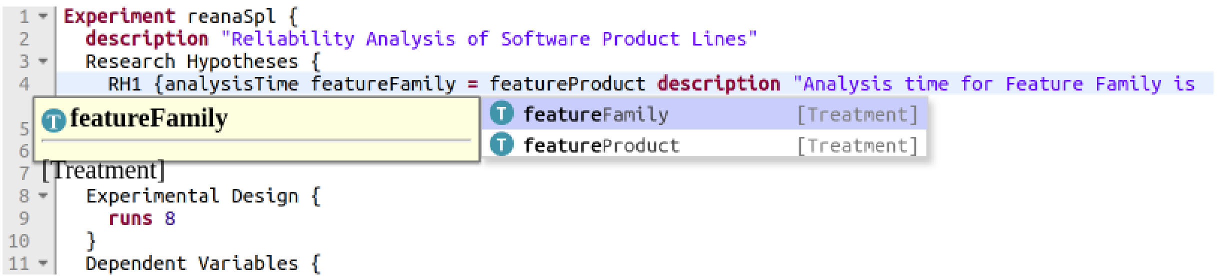
To conduct an experiment using our tool, an experimenter first must create an experiment specification using the DSL. To ease this task, we created a specific editor with syntax highlighting, content assist, syntax validation, static semantics validation, template proposals, and text hover (Figure 5.1). When an experiment is specified using the editor, its specification is type checked by the editor according to the grammar rules and additional static semantics validation rules. Each additional validation rule represents a static semantics non-conformity and can be reported as an error or as a warning by the editor.

Listing 5.1 shows a specification using the DSL, which was adapted from the original experiment conducted by Lanna et al. [2018]. In this specification, the research hypothesis RH1 (Line 4) compares the dependent variable analysisTime resulting of applying the treatments featureFamily and featureProduct. The dependent variable analysisTime (Line 10) has a corresponding instrumentation (Line 13). The instrumentation comprises 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 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 named argument and uses the execution reanaEvaluator (Lines 19 and 20). Each object defines a parameter named 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.

Listing 5.1: Example of an experiment specification

```
Experiment reanaSpl {
  description "Reliability Analysis of Software Product Lines"
  Research Hypotheses {
    RH1 {analysisTime featureFamily = featureProduct description
       "Analysis time for Feature Family is equal to  analysis time for
       Feature Product"}
  }
  Experimental Design {
    runs 8
  }
  Dependent Variables {
    analysisTime { description "Analysis time" scaleType Absolute unit
       "ms" instrument analysisTimeCommand }
  }
  Instruments{
    analysisTimeCommand {command  "/usr/bin/time -v"  valueExpression
       "Total analysis time:" }
  }
  Factors {
    strategy { description "Analysis Strategy" scaleType Nominal}
  }
  Treatments {
    featureFamily description "Feature Family"  factor strategy
       parameters{argument "FEATURE_FAMILY"} execution reanaEvaluator,
    featureProduct description "Feature Product"  factor strategy
       parameters{argument "FEATURE_PRODUCT"} execution reanaEvaluator
  }
  Objects  { description "SPL" scaleType Nominal {
        lift {
            description "Lift"  parameters {spl "lift"}
        },
        intercloud {
            description "Intercloud"  parameters {spl "intercloud"}
        }
    }
  }
  Executions {
    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_behavioral_model.xml"
34        }
35    }
36 }
```



```
Experiment reanaSpl {
  description "Reliability Analysis of Software Product Lines"
  Research Hypotheses {
    RH1 {analysisTime featureFamily = featureProduct description "Analysis time for Feature Family is
 featureFamily                     featureFamily          [Treatment]
                                   featureProduct         [Treatment]
[Treatment]
  Experimental Design {
    runs 8
  }
  Dependent Variables {
```

Figure 5.1: DSL Editor

After specifying the experiment, the experimenter can have the execution and analysis scripts generated by running the command Generate. The command Generate and Run (Figure 5.2) generates the scripts and then run them. The execution script is executed by the running infrastructure, and, during execution, the execution status is presented to the experimenter (Figure 5.3). 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 analysis using the command Run Analysis or perform additional analysis using the raw data and the scripts. The boxplot presented in Figure 5.4 corresponds to the analysis of RH1, which compares the analysis time of the treatments Feature Family and Feature Product (Line 4, Listing 5.1), for the experimental object Lift (Lines 23–24, Listing 5.1).

5.2 Architecture

The tool architecture is modular and extensible due to Eclipse's extension mechanism. The core component comprises the grammar, the validators, and interfaces to define generators, commands, and access to database (Figure 5.5). 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

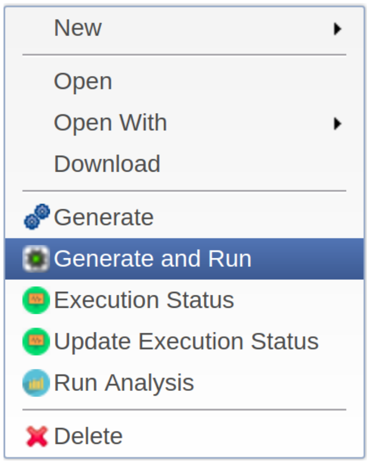
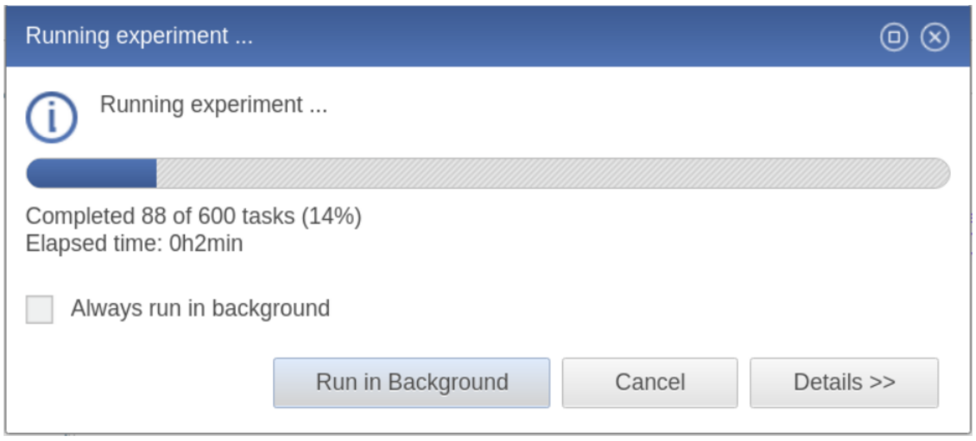


Figure 5.2: Generate and Run command


Running experiment ...

Running experiment ...

Completed 88 of 600 tasks (14%)
Elapsed time: 0h2min

Always run in background

Figure 5.3: Execution Status

the 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 have identified Dohko [Leite et al., 2017] as a potential autonomic solution to be used in the proposed solution because it not only fulfills all the requirements presented in Section 4.5 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 manually performing the configuration and initialization of the computing infrastructure with enough resources to run the experiments in a timely manner. In addition, Slurm [Yoo et al., 2003] 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

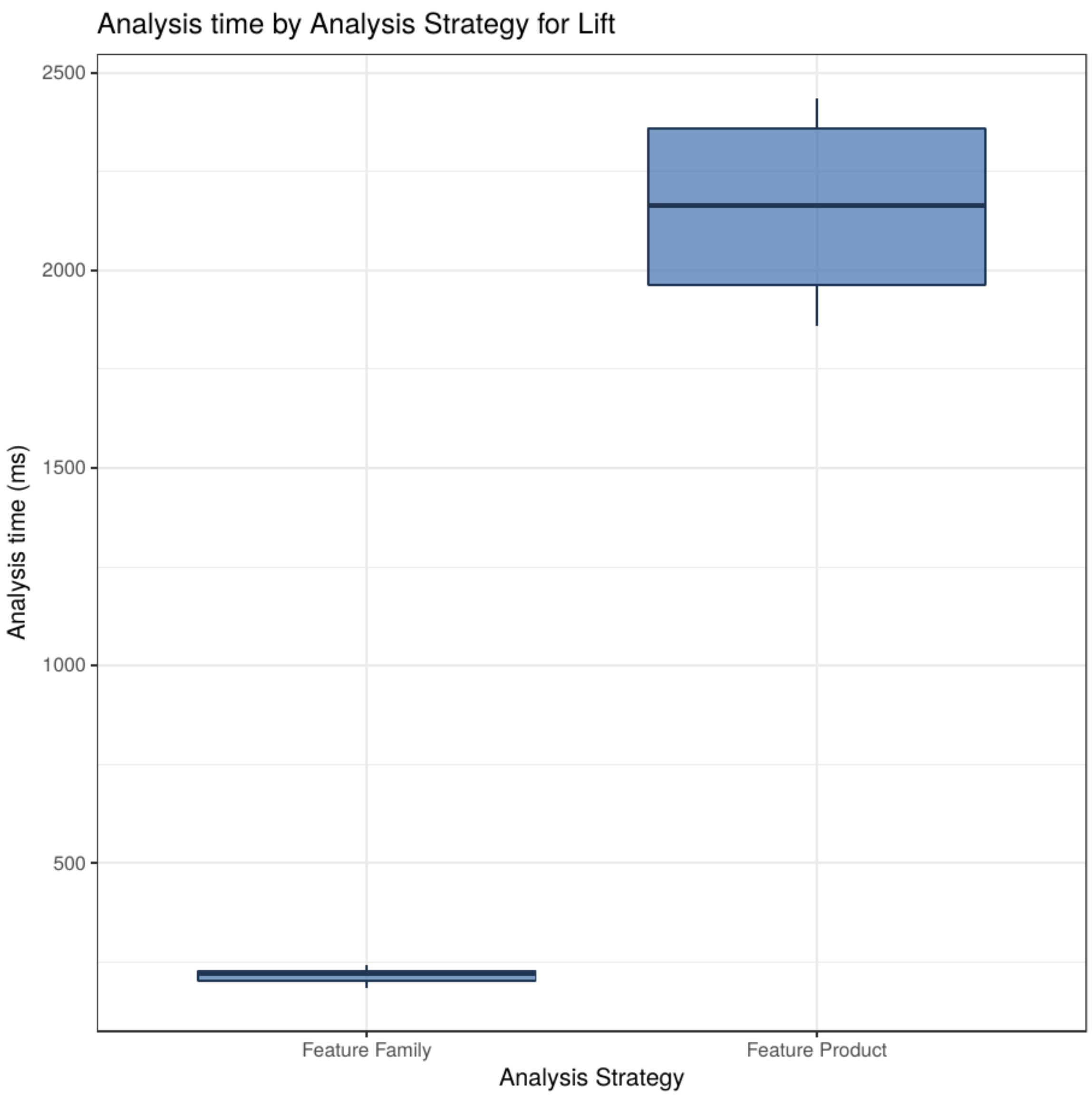


Figure 5.4: Excerpt of an Analysis Report

solution in our approach. However, we are using Dohko since it can manage resources not only in clusters but also in inter-cloud environments. Dohko was also integrated with *runexec* [Beyer et al., 2015] since it fulfills some requirements for reliable benchmarking and accurate resource measurements. To run analysis, we created an environment with R[1] for data analysis and Latex[2] for presentation of results. We actually run R Sweave scripts, which embed R code chunks in Latex documents. By doing so, we aim to achieve a Reproducible Research, as proposed by Madeyski and Kitchenham [2017].

Each main component, i.e., the supporting framework, the execution environment, the analysis environment, and the database, is run in its own Docker container, which enables

[1]https://www.r-project.org/
[2]https://www.latex-project.org/

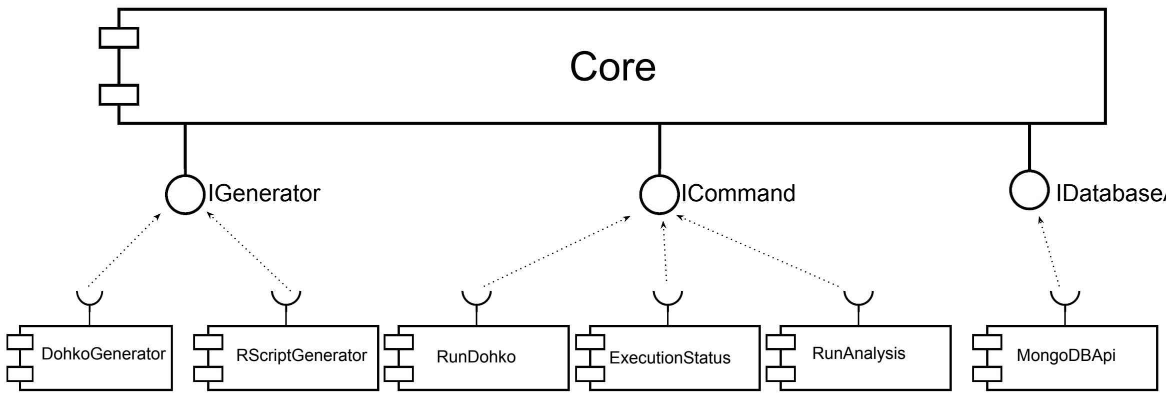


Figure 5.5: Tool Components

distributed execution, environment isolation, and portability. 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 the tool to be run in distinct environments, consequently, easing execution and replication of experiments.

5.3 Implementation

Using Xtext[3], we created the DSL partially based on ExpDSL Freire et al. [2013]. ExpDSL comprises four views: process view, metric view, experimental plan view, and questionnaire view. Metric view and experimental view are the same for human-oriented and technology-oriented experiments; thus, they can be reused in our work. Nevertheless, since the process view and the questionnaire view are bound to human-oriented experiments, they cannot be reused in our work. So, we created our DSL with new constructs for technology-oriented experiments, which enables 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 domain-specific language designed for the description of textual languages. Our full DSL grammar is presented in Appendix A. The parser of the DSL parses the specification of an experiment

[3]https://eclipse.org/Xtext/

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 using the Xtend[4] language. The validators complement the validations provided by the grammar rules to check the well-formedness (Definition 1) of the experiment specification. We created eight validation rules, four are reported as error and four are reported as warning. The validators are used to check the following non-conformities: if two distinct hypotheses perform the same treatments comparison (error); if a hypothesis compares a treatment with itself (error); if a hypothesis compares treatments from distinct factors (error); if a parameter used in a command line is invalid (error); if a dependent variable is never used (warning); if a factor is never used (warning); if a treatment is never used (warning); and if an execution is never used (warning). Our validation rules are listed in Appendix B.

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

Since we are using Dohko as infrastructure solution, the execution script is actually a Dohko Application Descriptor. Listing 5.2 is an excerpt of the generated execution script corresponding to the experiment specification in Listing 5.1. According to Algorithm 1, the execution script generator applies the treatments to the objects according to the design function. Currently, the tool 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 (Lines 6–8, Listing 5.1), 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 5.2). The command line of each application is generated by combining the instrumentation command and the execution command. In addition, 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 5.1) related to the dependent variable analysisTime, the command line defined for reanaEvaluator (Line 33, Listing 5.1), the parameter argument defined for the treatment featureFamily (Line 19, Listing 5.1), and the parameter spl defined for the object lift (Line 24, Listing 5.1). Finally, the resulting application is repeated the number of times defined by runs (Line 7, Listing 5.1). For the sake of brevity, we omitted these repetitions in Listing 5.2.

[4]http://www.eclipse.org/xtend/

Listing 5.2: Excerpt of a generated execution script corresponding to the experiment specification in Listing 5.1

```
---
name: "reanaSpl"
description: "Reliability Analysis of Software Product Lines"
blocks:
  - applications:
    - name: "featureFamily_lift_0"
      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_behavioral_model.xml"
  - applications:
    - name: "featureFamily_intercloud_0"
      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_behavioral_model.xml"
  - applications:
    - name: "featureProduct_lift_0"
      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_behavioral_model.xml"
  - applications:
    - name: "featureProduct_intercloud_0"
      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_behavioral_model.xml"
```

The corresponding generated analysis script is an R Sweave script (Listing 5.3). 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 using R code (Lines 6–24). First, a boxplot is generated (Lines 7–15) using 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 if 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 non-parametric test (Line 21) and then presents the results (Line 22).

Listing 5.3: 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}
```

Using DSLFORGE [Lajmi et al., 2014], an initial version of the supporting framework has been automatically generated from the DSL 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 version of the framework with additional commands to enable execution, monitoring, data analysis, and presentation of results.

The use of our DSL empowers researchers to specify experiments using experimentation concepts (e.g., experimental design, treatment, experimental object, dependent variable). The tool we created using Xtext and DSLFORGE supports the researcher in specifying the experiment by providing a specific editor with syntax highlighting, content assist, syntax validation, static semantics validation, template proposals, and text hover. A model-driven approach is 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. 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. The objective is to provide 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.

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 [Mattoso et al., 2010]. 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.

Chapter 6

Preliminary Evaluation

As a preliminary evaluation, the proposed solution was assessed with respect to automation, level of abstraction, and correctness. First, 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 4.3 and 4.4). Then, we investigated, in Section 6.1, the expressiveness of our tool to specify technology-oriented experiments (RQ 1) and if it can be used to enable sound automation of execution and analysis from the specification of technology-oriented experiments (RQ 2 and RQ 3). Finally, we evaluated the level of abstraction (Section 6.2) 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).

6.1 Execution and Analysis Automation

The main goal of this section is to assess the feasibility of our tool to provide automation in the experimentation process and is guided by the following research questions:

RQ 1. Is the DSL expressive enough to specify technology-oriented experiments?

RQ 2. Can the proposed tool be used to enable sound automation of execution from the specification of technology-oriented experiments?

RQ 3. Can the proposed tool be used to enable sound automation of analysis from the specification of technology-oriented experiments?

6.1.1 Evaluation Method

To address RQs 1 to 3, we first randomly selected three previously published experiments meeting the criteria described in Section 6.1.2. For each experiment, we performed two replications: one using our tool and another using the scripts provided by the authors. With our tool, we specified each experiment using our DSL, which assesses if the DSL is expressive enough to specify technology-oriented experiments (RQ 1). Since the main goal of the evaluation is to assess the feasibility of the tool, not the usability, we used the DSL ourselves (as future work, we plan an independent usability evaluation). Then, we used the tool to, from specification, generate and execute execution and analysis scripts. By doing so, we assess if the proposed tool can be used to enable sound automation of execution (RQ 2) and analysis (RQ 3) from the specification of technology-oriented experiments. However, execution and analysis must be sound. For this reason, we also replicated the experiments using original scripts, and, then, compared the results with the results obtained with our tool to assure that not only the tool can generate execution an 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 our proposal, we conducted external replications, with no interaction with original experimenters. We used the published papers and the lab 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 change how the experiment is executed and measured, 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 vCPUs 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 the repository located at https://github.com/eneiascs/dsm-experiments-evaluation/tree/dissertation.

6.1.2 Experiments Selection

We selected three experiments meeting the following criteria:

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

- The experiment is technology-oriented, i.e., 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 the selection criteria presented in above, we selected the following experiments:

Experiment 1. Bak and Duggirala [2017] 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)[1]. In their optimization evaluation, the authors examined the effects of optimizations for computing reachability for linear-time invariant systems with inputs. They compared the 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 references 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 the runtime, the number of steps in the problem was varied by changing the step size and keeping the time bound fixed at 2π. Then the runtime for each optimization was measured, recording 10 measurements in each case. The results are presented in Figure 7.1. The performance of the Basic algorithm (Basic) is improved by the 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) is a lower bound.

Experiment 2. Brennan et al. [2017] presented a constraint caching framework to expedite potentially expensive satisfiability and model-counting queries. Their techniques were implemented in a tool named Cashew[2], which was built as an extension of the Green caching framework [Visser et al., 2012]. Cashew was also integrated with Symbolic PathFinder (SPF) [Păsăreanu et al., 2013] and the ABC [Aydin et al., 2015] model-counting constraint

[1]http://stanleybak.com/papers/bak2017cav_repeatability.zip
[2]https://github.com/vlab-cs-ucsb/cashew/

solver. The authors investigated the effects of their normalization procedure on model-counting datasets of string constraints. Kaluza dataset [Saxena et al., 2010], 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 7.1. The results show that, on the SMC-Big set without duplicates, Cashew achieved a speedup over 10x, and, on the SMC-Small set without duplicates, 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 high number of 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 7.2 shows the number of orbits that are achieved by different subsets of the transformations. 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. [2017]. In this experiment, the authors investigated the effects of their normalization procedure on side-channel analysis. They used Symbolic PathFinder [Păsăreanu et al., 2013] with Cashew to symbolically execute four Java programs that operate on strings: Password1, Password2, Obscure, and CRIME. Password1 contains a method that checks whether or not a user-given string matches a secret password. Password2 is 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. [2014], which is a password change authorizer. CRIME is a Java version of a well-known attack, Compression Ratio Info-leak Made Easy [Bang et al., 2016; Rizzo, 2012]. For each of the four programs under analysis, they ran 1,000 symbolic-execution-based side-channel analyses, using as the secret each of the 1,000 passwords in the *RockYou1K* dataset [Weir et al., 2010]. Table 7.5 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 normalization achieved only 1.06x. The hit/miss ratios improve dramatically when switching to Cashew.

6.2 Level of Abstraction

The main goal of this section is to assess our tool with respect to the level of abstraction from the perspective of experimenters. This assessment is guided by the following research questions:

RQ 4. Does the proposed tool raise the level of abstraction required to execute and analyze a technology-oriented experiment?

RQ 5. What is the level of abstraction of the language constructs?

6.2.1 Evaluation Method

To address RQ 4, we compared the level of abstraction of original specifications with specifications using the DSL of experiments used in Section 6.1. 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 does it; abstraction is a process of generalization, eliminating detail, removing inessential information [Ward, 1995].
- **Number of potential implementations:** abstract specifications have more potential implementations, whereas moving to a lower level means restricting the number of potential implementations [Ward, 1995].
- **Domain concepts:** DSM raises the level of abstraction beyond general purpose languages by specifying the solution directly using problem domain concepts [Kelly and Tolvanen, 2008].
- **Complexity:** DSM reduces complexity, since the language deals only with high-level domain concepts, and all details of implementation are hidden in code generators [Kelly and Tolvanen, 2008].

To address RQ 5, we first selected well-established guides in Software Engineering experimentation and then compared their key concepts with DSL constructs/elements. Then, we classified the DSL 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 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.

Chapter 7

Results and Analysis

We present and discuss the results of the empirical evaluation we performed regarding the use of the proposed solution to provide automation of execution and analysis (Section 7.1) and to raise the level of abstraction (Section 7.2) in the experimentation process. We also discuss the lessons learned (Section 7.3), and the threats to validity (Section 7.4).

7.1 Execution and Analysis Automation

We present the results of replicating Experiments 1 to 3.

The first replicated experiment was Experiment 1. The results of replicating the experiment with the tool (Figure 7.2b) are consistent with the replication using original scripts (Figure 7.2a) and with the results presented in the paper (Figure 7.1): Basic is the worst optimization, followed by Warm and Decomp; Hylaa is better than Decomp; and NoInput is a lower bound. The relative differences between the results with and without the tool are presented in Figure 7.3. The differences are really high for runtime values below one second, reaching more than 70% for NoInput. However, the differences decrease quickly to nearly 20% for one second, to 10% for two seconds, and to 5% for three seconds. Above three seconds, the differences keep below 5%.

The second replicated experiment corresponds to Experiment 2. Due to the high number of duplicates present in original dataset and to avoid an excessive time-consuming experiment, in our replications we used only the Kaluza dataset without duplicates. The results are presented in Tables 7.3 and 7.4. Using original scripts, Cashew achieved a speedup of 20.62x on the SMC-Big, and 2.43x on SMC-Small. Using our tool, 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 7.1 and 7.2). The only difference in results is the number of orbits on SMC-Small with no transformation,

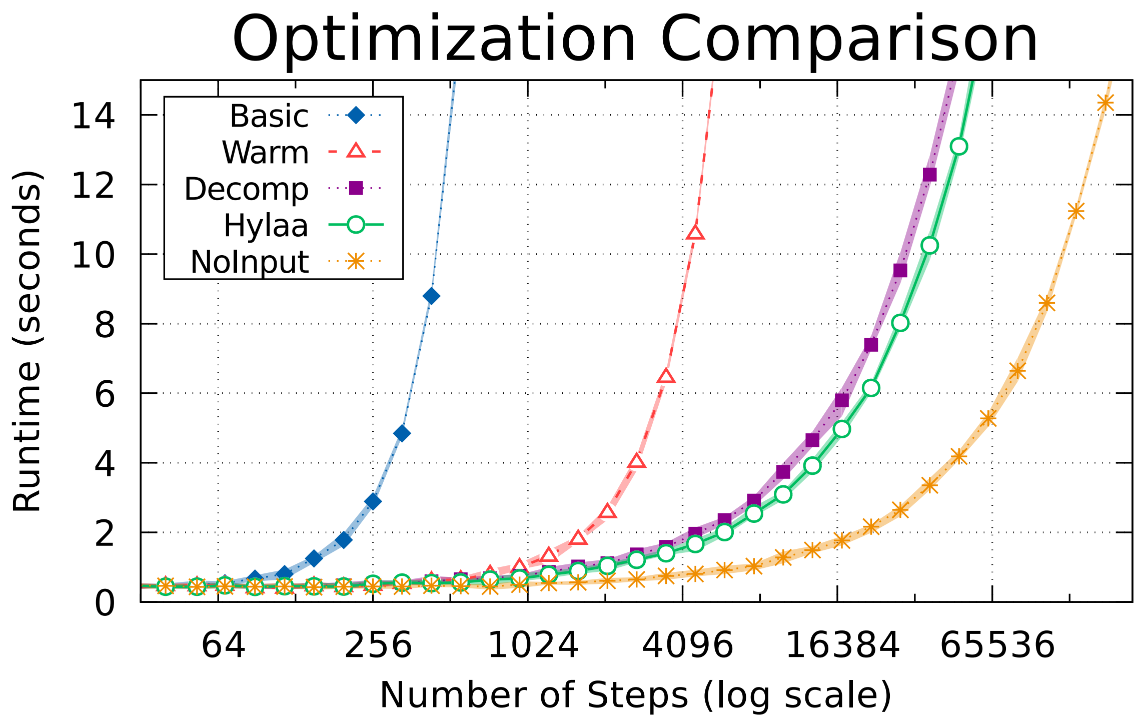
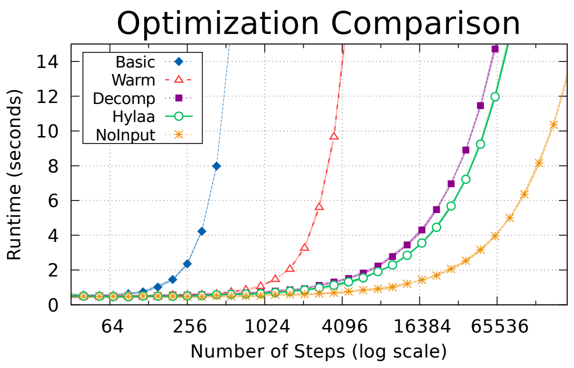
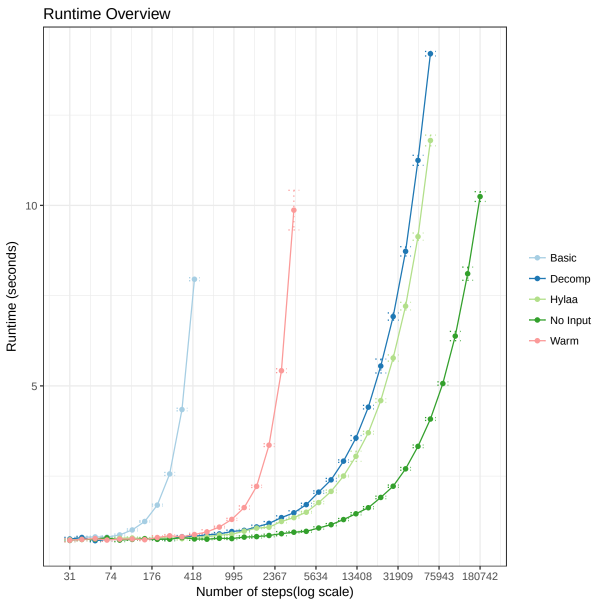


Figure 7.1: Results of Optimization Comparison [Bak and Duggirala, 2017]



(a) Original scripts



(b) Tool

Figure 7.2: Results of replicating Experiment 1

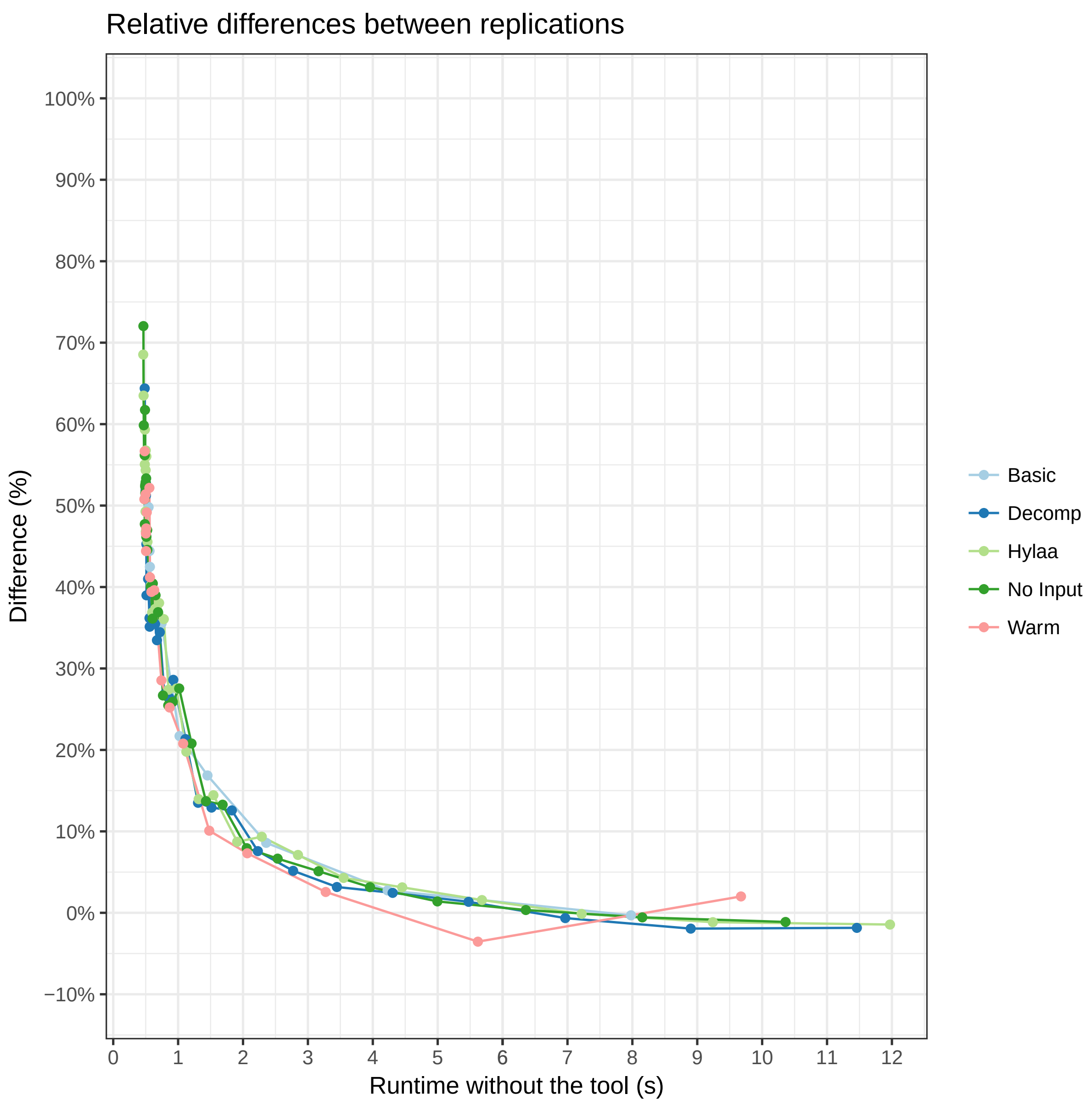


Figure 7.3: Relative differences between replications with and without the tool for Experiment 1

Table 7.1: Results of Model counting SMC-Big and SMC-Small [Brennan et al., 2017]

		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 7.2: Effect of transformations on orbit refinement [Brennan et al., 2017]

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

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 and without the tool are presented in Figure 7.4. 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 7.6 and 7.7. Cashew achieved an average speedup of 2.8x (original) and 2.43x (tool), while caching without normalization achieved 1.07x (original) and 1.08x (tool). 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 paper (Table 7.5). However, there is a difference in results for Obscure in relation to the number of hits and misses. This discrepancy is likely due to changes in the version of ABC. Nevertheless, a further investigation should be

Table 7.3: Results of Model counting SMC-Big and SMC-Small (replication)

		Original scripts			Tool		
		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 7.4: Effect of transformations on orbit refinement (replication)

	Original scripts		Tool	
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

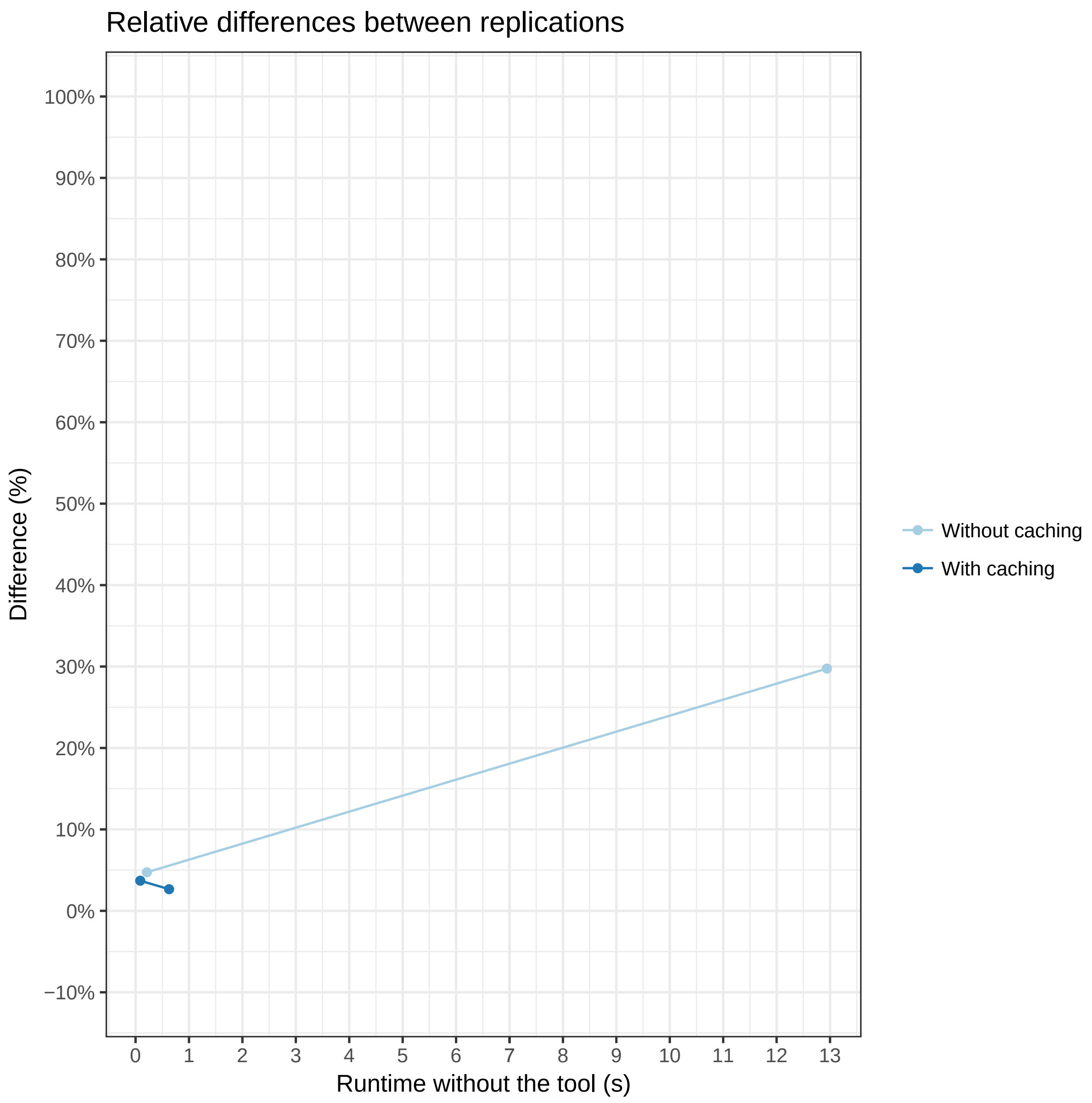


Figure 7.4: Relative differences between replications with and without the tool for Experiment 2

Table 7.5: Results of SPF-based quantitative analyses of string programs [Brennan et al., 2017]

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 7.6: 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

carried out to confirm or refute this hypothesis. The relative differences between the total runtime results with and without the tool are presented in Figure 7.5. The differences are below 5% for all cases.

Since we could specify three experiments, this suggests that the DSL is expressive enough to specify technology-oriented experiments (RQ 1). From experiment specifications, we used the tool to automatically generate execution and analysis scripts. Then, we used the tool to execute the execution scripts and collect the results. Finally, we used the tool to analyze the execution results using the previously generated analysis scripts.

When it comes to the execution results, there were some differences in relation to runtime. For Experiment 1, the differences were higher for lower runtime values (below

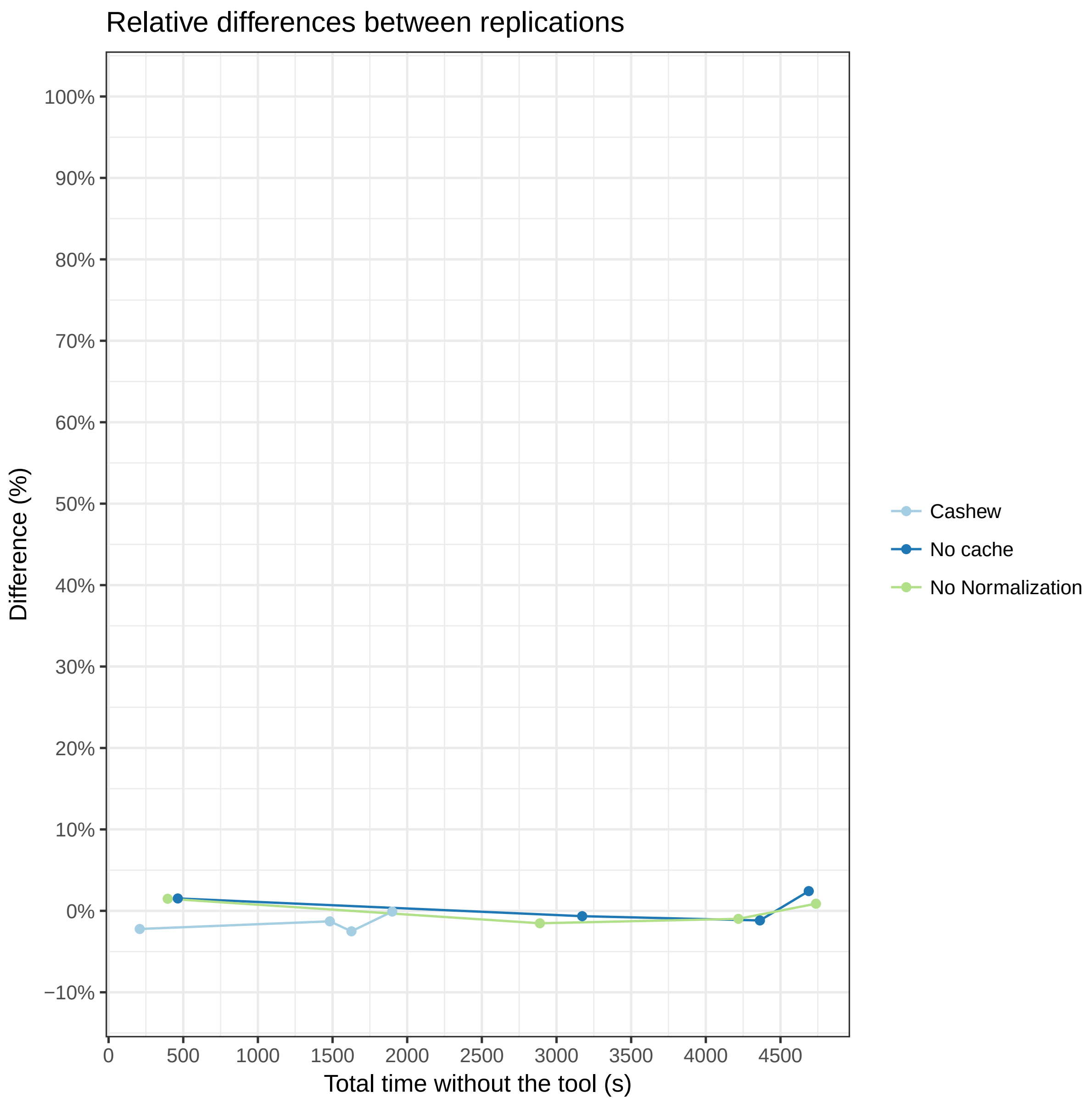


Figure 7.5: Relative differences between replications with and without the tool for Experiment 3

Table 7.7: Results of SPF-based quantitative analyses of string programs (tool)

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

one second), but the differences decrease quickly for higher runtime values (above three seconds). For Experiment 2, there are only two treatments and two objects; thus, only four data points. For SMC-Big without caching, the difference is around 30%, and the runtime without the tool is 12.94s. For the other cases, the differences are below 5%, and the runtime values without the tool are 0.21s, 0.22s, and 0.63s. For Experiment 3, the total runtime is higher, from near 100 seconds to almost 5000 seconds, and the differences are all less than 5%. These preliminary results suggest that the overhead of the tool is more significant for lower runtime values (below one second), although the results for Experiment 2 diverge from this hypothesis. For this reason, further experiments should be conducted to thoroughly investigate this issue. However, when we evaluated other dependent variables that not depend on the execution environment, such as number of orbits, number of hits, and number of misses in caching systems, the execution results were exactly the same with and without the tool.

Although there are some differences regarding execution time between the replications with and without the tool, the qualitative results are consistent and lead to the same conclusions. Thus, these preliminary results suggest that not only the proposed solution can be used to enable 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).

7.2 Level of Abstraction

We present the results of comparing the level of abstraction of original specifications with specifications using the DSL of experiments used in Section 6.1 (Section 7.2.1) and also the results of comparing the DSL constructs/elements with key experimentation concepts (Section 7.2.2).

7.2.1 Experiment Specifications

For execution purposes, the authors of Experiment 1 created an execution script (Listing 2.1) and a Gnuplot configuration file (Listing 2.3). Instead, we created a corresponding specification using the DSL (Listing 7.1).

The execution scripts and experiment specifications of Experiments 2 and 3 are presented in Listings D.1 to D.4.

Listing 7.1: Excerpt of an experiment specification

```
Experiment hylaaOptimization {
  Research Hypotheses {
    RH1 {time Hylaa = Warm description "Runtime time for Hylaa is equal
        to  runtime time for Warm" },
    RH2 {time Hylaa = Decomp description "Runtime time for Hylaa is
        equal to  runtime time for Decomp"},
    RH3 {time Hylaa = Basic description "Runtime time for Hylaa is
        equal to  runtime time for Basic"},
    RH4 {time Hylaa = NoInput description "Runtime time for Hylaa is
        equal to  runtime time for NoInput"}
  }
  Experimental Design {
    runs 10
  }
  Dependent Variables {
    time { description "Runtime" scaleType Absolute unit "seconds"
        instrument timeInstrument }
  }
  Instruments {
    timeInstrument {command  "/usr/bin/python -u
        /opt/optimizations/time.py"  valueExpression "runtime:"}
  }
  Factors {
    optimization { description "Optimization" scaleType Nominal}
  }
  Treatments {
    Hylaa description "Hylaa"   factor optimization parameters {params
        ""}  execution hylaaTool,
    Warm description "Warm" factor optimization parameters {params
        "settings.opt_decompose_lp=False"} execution hylaaTool,
    Decomp description "Decomp" factor optimization parameters {params "settings.opt_warm_start_lp=False"} execution hylaaTool,
    Basic description "Basic" factor optimization parameters {params "settings.opt_warm_start_lp=False settings.opt_decompose_lp=False"} execution hylaaTool,
    NoInput description "No Input" factor optimization parameters {params ""} execution hylaaToolNoInput
  }
  Objects {description "Number of steps" scaleType Logarithmic {
      steps31 {description "31 steps" value "31"  parameters {num_steps "31", step_size "0.200000000"}},
      steps40 {description "40 steps" value "40" parameters {num_steps "40", step_size "0.153846154"}},
      steps53 {description "53 steps" value "53" parameters {num_steps "53", step_size "0.118343195"}},
      steps106948 {description "106948 steps" value "106948" parameters {num_steps "106948", step_size "0.000058720"}},
      steps139032 {description "139032 steps" value "139032" parameters {num_steps "139032", step_size "0.000045169"}},
      steps180742 {description "180742 steps" value "180742" parameters {num_steps "180742", step_size "0.000034746"}}
    }
  }
  Executions {
    hylaaTool {
      command "/usr/bin/python -u /opt/hyst-1.5/src/hybridpy/hybridpy/tool_hylaa.pyc ${treatment.name}/${object.parameter.num_steps}.py -"
      timeout 15
      preprocessing {
        mkdir{command "mkdir -p ${treatment.name}"},
    hyst{command "java -jar /opt/hyst-1.5/src/Hyst.jar -i /opt/optimizations/io.xml -o ${treatment.name}/${object.parameter.num_steps}.py -tool hylaa '-settings settings.print_output=False ${treatment.parameter.params} -step ${object.parameter.step_size}'"}
      }
      postprocessing {
        rm{command "rm -f ${treatment.name}/${object.parameter.num_steps}.py"}
      }
   },
    hylaaToolNoInput {
```

```
49        command "/usr/bin/python -u
              /opt/hyst-1.5/src/hybridpy/hybridpy/tool_hylaa.pyc
              ${treatment.name}/${object.parameter.num_steps}.py -"
50        timeout 15
51        preprocessing {
52          mkdir{command "mkdir -p ${treatment.name}"},
53          hyst{command "java -jar /opt/hyst-1.5/src/Hyst.jar -i
                /opt/optimizations/ha.xml -o
                ${treatment.name}/${object.parameter.num_steps}.py -tool
                hylaa '-settings settings.print_output=False
                ${treatment.parameter.params} -step
                ${object.parameter.step_size}'"}
54        }
55
56        postprocessing {
57          rm{command "rm -f
                ${treatment.name}/${object.parameter.num_steps}.py"}
58        }
59      }
60    }
61  }
```

We present a comparison between the level of abstraction of original specifications with specifications using the DSL of Experiments 1 to 3 based on the criteria defined in Section 6.2.1:

- **Level of detail:** Since the DSL is declarative, it says only 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 the DSL, an experimenter needs only to specify the experimental design (Lines 8–10), treatments (Lines 20–26), and objects (Lines 27–34). 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 2.1), one must write not only the treatments and objects definitions but also the mechanics of applying the treatments to the objects (Lines 21–36).

- **Number of potential implementations:** The DSL is implemented by 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 Pyhton, one would have to use distinct implementations of the Python compiler, which, indeed, limits the potential implementations.

- **Domain concepts:** Our proposed DSL was created to be used in the Experimentation Domain. So, naturally, it uses domain concepts, such as Research Hypothesis (Line 2), Dependent Variables (Line 11), Treatments (Line 20), Objects (Line 27), etc. Unlike the DSL, the original scripts were created using Python, which is a general purpose language and does not contain any concept of the Experimentation Domain.

- **Complexity:** Since the DSL 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. On the other hand, to create execution scripts in Python, or any other imperative language, the experimenter must deal with the complexity of control flow statements, variable declarations, and so on. For instance, in Listing 2.1, to repeat the execution a number of times, first, the variable `num_trials` is declared (Line 10). Then, a loop control is used to repeat the execution the number of times defined in `num_trials` (Lines 31–32). Using the DSL, the experimenter simply defines the number of runs (Line 9).

Based on this comparison, we conclude that the proposed tool raises the level of abstraction required to execute and analyze a technology-oriented experiment (RQ 4).

7.2.2 DSL Constructs

We also present a comparison between DSL constructs and domain concepts. In this comparison, we considered all types defined in the DSL grammar. Based on the criteria defined in Section 6.2.1, we classified the grammar constructs in three groups: high-level constructs (Table 7.8), mid-level constructs (Table 7.9), and low-level constructs (Table 7.10).

As a result of the evaluation (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%). Despite the 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 the DSL. 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

Table 7.8: Comparison between DSL constructs and Domain Concepts (high-level constructs)

DSL Construct	Domain Concept		
	Jedlitschka et al. [2008]	Wohlin et al. [2012]	Juristo and Moreno [2013]
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 7.9: DSL mid-level constructs

DSL 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 7.10: DSL low-level constructs

DSL 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

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.

7.3 Discussion and Lessons Learned

We propose a DSM approach supporting technology-oriented experiments. The proposed solution was evaluated with respect to automation, level of abstraction, and correctness.

Automation. We used a model-driven approach to generate execution and analysis scripts from experiment specifications. This enables full automation of execution and analysis, and, thus, frees the researcher of the task of manually creating execution and analysis scripts, which could be error-prone, time-consuming, and requires knowledge on general purpose languages and statistics. By replicating three published experiments, we show that the DSL is expressive enough to specify technology-oriented experiments (RQ 1) and that the proposed tool can be used to enable sound automation of execution (RQ 2) and analysis (RQ 3) from the specification of technology-oriented experiments.

Abstraction. By creating a DSL using experimentation concepts, we raised the level of abstraction of experiments specifications. Although the experimenter must learn a new language, this language has a higher level of abstraction in relation to general purpose languages (RQ 4) since less detail must be provided in the specification, there are more potential implementations, the DSL uses domain concepts, and the DSL is less complex. By comparing the DSL constructs with domain concepts (RQ 5), we found that 54.35% are high-level constructs, 15.22% are mid-level constructs, and 30.43% are low-level constructs. Even the low-level constructs are less complex than general purpose language statements since they contain only declarative statements instead of control flow ones. In addition, the results suggest that, since the low-level constructs are related to the infrastructure, they could be moved from the DSL to the supporting framework.

Correctness. To assure the correctness of the results provided by our model, we defined some key correctness properties (Chapter 4). These correctness properties were formally proved, which assures that the results are consistent with the experiment specification.

7.4 Threats to Validity

The evaluation of automation (Section 6.1) is a quantitative evaluation based on replications. On the other hand, the evaluation of the level of abstraction (Section 6.2) is an analytical comparison. For both evaluations, we present the threats to validity:

Conclusion validity. To perform the replications with and without the tool, we used procedures and scripts as similar as possible to that presented by the authors in 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 results between the executions with and without the tool. In Experiment 1, each treatment is 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 enough to perform a significance test. It requires all the single measurements, or, at least, the mean and the variance of the sample [Box et al., 2005; Juristo and Moreno, 2013]. In Experiments 2 and 3, since each object is, in fact, a whole dataset, each treatment is 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. In addition, to mitigate the threat of using a bad instrumentation, in the replications, we used the same instrumentation used in the original experiments.

Internal validity. The measurement of performance, specially runtime, is sensibly affected by the execution environment. Other processes running in the same machine and consuming resources, such as cpu, memory, and disk access, may cause variations in the measured runtime. This could affect the comparison of the results of replications with and without the tool. 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 configurations to keep the execution environments as similar as possible.

Construction 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 references from the literature.

External validity. To empirically evaluate the proposed solution, we replicated distinct experiments from the automatic verification domain. To find technology-oriented experiments with the replication completely documented and all the artifacts available (Section 6.1.2), we direct our search to venues explicitly requiring reproducibility as part of the evaluation process or distinguishing it in accepted papers, and also to experiments more related to our research group, which may have restricted the domain of the experiments. 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 validity. We conducted the evaluation ourselves, which can introduce bias in the evaluation. 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.

Chapter 8

Conclusion

We presented a Domain-Specific Modeling Approach Supporting technology-oriented experiments. The solution comprises a DSL, execution and analysis script generators, a supporting framework, and a running infrastructure. All these components are integrated in a Web-based tool that implements the DSM approach, providing a means to specify runnable specifications at a high level of abstraction; automated execution, data analysis, and results presentation.

We empirically evaluated the practical applicability of the tool to provide automation in the experimentation process and its level of abstraction. The results suggest that the DSL is expressive enough to specify technology-oriented experiments and that the proposed tool can be used to enable sound automation of execution and analysis from the specification of technology-oriented experiments. In addition, the empirical assessment also suggests that the use of the DSL raises the level of abstraction of experiment specifications when comparing to general purpose languages. When it comes to the language constructs, the comparison with domain concepts shows that 54.35% are high-level constructs, 15.22% are mid-level constructs, and 30.43% are low-level constructs. 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 presented a formal model of the tool and some key correctness properties. These correctness properties were formally proved, which assures that the results are consistent with the experiment specification. Overall, we believe our DSM solution and supporting tool are a step towards improved efficiency of the experimentation process and correctness of its results.

8.1 Limitations

Although the DSL is expressive enough to specify technology-oriented experiments and the proposed tool can be used to enable automation of execution and analysis of technology-oriented experiments, there are some limitations.

Experimental Design. The experimental design applies only a (subset of) Cartesian product to relate treatments and experimental 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 scalability evaluations.

Experimental Objects. The tool is able to apply a treatment to an object the number of times defined by the experimenter. However, the object must be exactly the same. In some experiments [Beyer et al., 2018; Devroey et al., 2017], 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.

Output checking. Using our tool, 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. [2018].

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. If additional designs were added to the specification, the corresponding statistical test should also be added to analysis.

Evaluation. We evaluated neither the cost of learning the DSL nor its usability.

Manual Tasks. Although our solution can be used to enable automation of execution and analysis, the experimenter still has to perform some manual tasks, such as interpreting the results, drawing the conclusions, writing replication instructions, and publishing the lab package. In addition, a system administrator has to properly configure the running infrastructure to run the experiment. Then, the system administrator can publish a Docker image with these configurations so that other researchers can replicate the experiment or conduct further analyses.

8.2 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 runnable 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. [2015] 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* [Leite et al., 2017] so that our execution environment meets the requirements presentend by the authors.

Hauck et al. [2014] 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. Ginpex could be used, for instance, to evaluate the overhead of our tool and the running infrastructure.

Wang et al. [2005] 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. [2013] 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. [2008] presented an experimental Software Engineering Environment (eSEE) to support large-scale experimentation and scientific knowledge management 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. Their proposal includes definition, planning, execution, and packaging of primary and secondary studies. However, it does not support automated execution and data analysis from the experiment specification for technology-oriented experiments.

Arisholm et al. [2002] 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. [2008] 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.

Data analysis and presentation: Madeyski and Kitchenham [2017] 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 extend 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 before, although the aforementioned techniques help in conducting controlled experiments, they have a different and broad perspective and can be seen as complementary works.

8.3 Future Work

Indeed, as a preliminary contribution, our DSM solution has a number limitations (Section 8.1). Accordingly, in future work, some improvements could be made:

Experimental Design. 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.

Experimental Objects. Support the definition of related experimental objects as a single dataset so that the results can be analyzed as a single experimental object. This would enable the use of the tool in experiments where 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.

Output checking. Provide means to specify an output reference to check the actual output of execution. Currently, we run the tool specified by the experimenter and measure the dependent variable using the corresponding instrumentation. However, the experimenter must be assured that the tool related to the treatment is performing the work it is supposed to do rather than performing some arbitrary processing.

Analysis. 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.

Evaluation. 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, conduct further experiments to investigate the overhead of the tool in relation to runtime. The preliminary results suggest that the differences in runtime with and without the tool vary for distinct time ranges. However, this should be thoroughly investigated with additional experiments. Furthermore, evaluate additional aspects of the DSL, such as usability and the cost of learning the language by independent users. 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.

References

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.

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.

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.

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.

Jerry Banks. Introduction to simulation. In *Proceedings of the 1999 Winter Simulation Conference*, pages 7–13, 1999.

Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In *Model Checking Software*, pages 160–178. Springer, 2015.

Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on smt-based software verification. *Journal of Automated Reasoning*, 60(3):299–335, 2018.

Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

George EP Box, J Stuart Hunter, and William Gordon Hunter. *Statistics for experimenters: design, innovation, and discovery*, volume 2. Wiley-Interscience New York, 2005.

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.

Xiaoling Chen and Jeffrey T Chang. Planning bioinformatics workflows using an expert system. *Bioinformatics*, page btw817, 2017.

Minh Thanh Chung, Nguyen Quang-Hung, Manh-Thin Nguyen, and Nam Thoai. Using docker in high performance computing applications. In *Communications and Electronics (ICCE), 2016 IEEE Sixth International Conference on*, pages 52–57. IEEE, 2016.

Marcus Ciolkowski. *An Approach for quantitative aggregation of evidence from controlled experiments in software engineering.* Fraunhofer Verlag, 2012.

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 *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 424–429. IEEE, 2017.

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. Springer, 2008.

Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

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.

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.

Omar S Gómez, Natalia Juristo, and Sira Vegas. Replications types in experimental disciplines. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 3, 2010.

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.

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.

Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. 2001.

Claudia Houben and Alexei A Lapkin. Automatic discovery and optimization of chemical processes. *Current Opinion in Chemical Engineering*, 9:1–7, 2015.

Markus C Huebscher and Julie A McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3):7, 2008.

Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting experiments in software engineering. In *Guide to advanced empirical software engineering*, pages 201–228. Springer, 2008.

Natalia Juristo and Ana M Moreno. *Basics of software engineering experimentation.* Springer Science & Business Media, 2013.

Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, 2008.

Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

Jelena Kovacevic. How to encourage and publish reproducible research. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–1273. IEEE, 2007.

Amine Lajmi, Jabier Martinez, and Tewfik Ziadi. Dslforge: Textual modeling on the web. *DemosMoDELS*, 1255, 2014.

Leslie Lamport. Latex: A document preparation system. 1994.

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.

Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat*, pages 575–580. Springer, 2002.

Alessandro Ferreira Leite, Vander Alves, Genaína Nunes Rodrigues, Claude Tadonki, Christine Eisenbeis, and Alba Cristina Magalhaes Alves De Melo. Autonomic provisioning, configuration, and management of inter-cloud environments based on a software product line engineering method. In *International Conference on Cloud and Autonomic Computing*, pages 72–83, 2016.

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.

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.

Lech Madeyski and Barbara Kitchenham. Would wider adoption of reproducible research be beneficial for empirical software engineering research? *Journal of Intelligent & Fuzzy Systems*, (Preprint):1–13, 2017.

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.

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.

Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys*, 37(4):316–344, 2005.

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.

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.

Konstantinos Plakidas, Srdjan Stevanetic, Daniel Schall, Tudor B Ionescu, and Uwe Zdun. How do software ecosystems evolve? a quantitative assessment of the r ecosystem. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 89–98. ACM, 2016.

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.

T. Rizzo, J. Duong. The crime attack. *Ekoparty Security Conference*, 2012.

JarrettJarrett Rosenberg Rosenberg. Statistical methods and measurement. In *Guide to Advanced Empirical Software Engineering*, pages 155–184. Springer, 2008.

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.

Forrest J Shull, Jeffrey C Carver, Sira Vegas, and Natalia Juristo. The role of replications in empirical software engineering. *Empirical software engineering*, 13(2):211–218, 2008.

Janice Singer, Susan E Sim, and Timothy C Lethbridge. Software engineering data collection for field studies. In *Guide to Advanced Empirical Software Engineering*, pages 9–34. Springer, 2008.

Mirko Sonntag, Dimka Karastoyanova, and Frank Leymann. The missing features of workflow systems for scientific computations. In *Software Engineering*, pages 209–216, 2010.

Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

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.

R Core Team. R core team. r: A language and environment for statistical computing. r foundation for statistical computing, vienna, austria. Relatório técnico, Software Vienna, Austria: R Foundation for Statistical Computing, 2018.

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.

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.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 2000.

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *SIGSOFT FSE*, 2012.

Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages.* 2013. URL dslbook.org.

Yanyan Wang, Matthew J Rutherford, Antonio Carzaniga, and Alexander L Wolf. Automating experimentation on distributed testbeds. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 164–173. ACM, 2005.

Martin Ward. A definition of abstraction. *Journal of Software: Evolution and Process*, 7 (6):443–450, 1995.

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.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering.* Springer Science & Business Media, 2012.

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.

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.

Appendices

Appendix A

DSL Grammar

In this appendix, we present the grammar of the DSL (Listing A.1) created using Xtext.

Listing A.1: DSL Grammar

```
grammar br.unb.autoexp.AutoExp with org.eclipse.xtext.common.Terminals
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate autoExp "http://www.unb.br/autoexp/AutoExp"

Model:
    experiments+=Experiment*
;

Experiment returns Experiment:
    'Experiment'
    name=ID
    '{'
    ('Authors' '{' authors+=Author ("," authors+=Author)* '}')?
    ('description' description=STRING)?
    ('Abstract' abstract=Abstract)?
    ('Keywords' '{' keywords+=Keyword ("," keywords+=Keyword)* '}')?
    ('Goals' '{' goals+=Goal ("," goals+=Goal)* '}')?
    ('Research Questions' '{' researchQuestions+=ResearchQuestion (","
       researchQuestions+=ResearchQuestion)* '}')?
    ('Research Hypotheses' '{' researchHypotheses+=ResearchHypothesis
       ("," researchHypotheses+=ResearchHypothesis)* '}')?
    ('Threats' '{' threats+=Threat ("," threats+=Threat)* '}')?
    'Experimental Design' experimentalDesign=ExperimentalDesign
    'Dependent Variables' '{' dependentVariables+=DependentVariable
       ("," dependentVariables+=DependentVariable)* '}'
    ('Instruments' '{' instruments+=Instrument (","
       instruments+=Instrument)* '}')?
    'Factors' '{' factors+=Factor ("," factors+=Factor)* '}'
    'Treatments' '{' treatments+=Treatment ("," treatments+=Treatment)*
       '}'
    ('Groups' '{' groups+=ObjectGroup ("," groups+=ObjectGroup)* '}')?
    'Objects' '{' 'description' objectsDescription=STRING 'scaleType'
       objectsScaleType=ScaleType '{'
       experimentalObjects+=ExperimentalObject (","
       experimentalObjects+=ExperimentalObject)* '}' '}'
    'Executions' '{' executions+=Execution ("," executions+=Execution)*
       '}'
    ('Analysis' analysis=Analysis)?
    'Infrastructure' infrastructure=Infrastructure
    '}';

Infrastructure:
    {Infrastructure}
    '{'
        user=User
        ('requirements' requirements=Requirements)?
        ('preconditions' preconditions=Preconditions)?
        ('clouds' '{' clouds+=Cloud  (',' clouds+=Cloud)* '}')?
        ('on-finish' onFinish=OnFinishType)?
    '}'
;

Preconditions:
    {Preconditions}
    '{'
     (packages+=STRING   (',' packages+=STRING)*)?
    '}'
;
User:
    'user' '{'
     'username' username=STRING
     ('keys' '{' keys+=UserKey   (',' keys+=UserKey)* '}')?

     '}'
;
UserKey:
    name=STRING
    ('{'
     ('privateKey' privateKey=STRING)?
     ('publicKey' publicKey=STRING)?
     ('fingerprint' fingerprint=STRING)?
    '}')?
    ;
Requirements:
    '{'
    'cpu' cpu=INT
    'memory' memory=INT
    'platform' platform=PlatformType
    'cost' cost=BigDecimalType
    'number-of-instances-per-cloud' instancesPerCloud=INT
    '}'
;
PlatformType:
    typeName=('LINUX' | 'WINDOWS' );
BigDecimalType returns ecore::EBigDecimal:
 INT ('.' INT)?;

Cloud:
    name=STRING
    '{'
     provider=CloudProvider
     accessKey=AccessKey
    ('regions' '{' regions+=Region  (',' regions+=Region)*'}')?
    ('instanceTypes' '{' instanceTypes+=InstanceType  (','
       instanceTypes+=InstanceType)*'}')?
    '}'
    ;

CloudProvider:
    'provider' name=STRING
    ('{'
    ('maxResourcePerType' maxResourcePerType=INT)?
    ('description' description=STRING)?
     ('serviceClass' serviceClass=STRING)?
     '}')?

;

InstanceType:
    {InstanceType}
    name=STRING
    ('instances' numberOfInstances=INT)?
;

Region:
    name=STRING
    ('{'
    ('endpoint' endpoint=STRING)?
    ('status' status=StatusType)?
    ('city' city=STRING)?
    ('geographicRegion' geographicRegion=INT)?
    ('zones' '{' zones+=Zone  (',' zones+=Zone)*'}')?
    '}')?
;

StatusType:
    typeName=('UP' | 'DOWN' );

Zone:
    name=STRING
    (status=STRING)?
;
AccessKey:
    'access-key' accessKey=STRING
    'secret-key'secretKey=STRING
;
OnFinishType:
    typeName=('NONE' | 'SHUTDOWN' |'TERMINATE' );
Abstract returns Abstract:
    Abstract_Impl | SimpleAbstract | StructuredAbstract;

Goal returns Goal:
    Goal_Impl | SimpleGoal | StructuredGoal;

ExperimentalDesign returns ExperimentalDesign:
    '{'
    ('type' type=DesignType)?
    'runs' runs=INT

    ('Restrictions' '{' restrictions+=Restriction (","
       restrictions+=Restriction)* '}')?

    ('Context Variables' '{' contextVariables+=ContextVariable (","
       contextVariables+=ContextVariable)* '}')?
    '}';

Restriction returns Restriction:
    treatment=[Treatment|ID] 'objects' '{'
       objects+=[ExperimentalObject|ID] (","
       objects+=[ExperimentalObject|ID])* '}'
;

```

```
Execution returns Execution:
    name=ID
    '{'
    ('command' cmd=STRING)?
    ('timeout' timeout=BigDecimalType)?
    ('preconditions' preconditions=Preconditions)?
    ('result' result=File)?
    ('files' '{' files+=File ("," files+=File)* '}')?
    ('preprocessing' '{' preProcessingExecutions+=Execution (","
       preProcessingExecutions+=Execution)* '}')?
    ('postprocessing' '{' postProcessingExecutions+=Execution (","
       postProcessingExecutions+=Execution)* '}')?
    '}';

Analysis returns Analysis:
    {Analysis}
    name=ID
    '{'
    ('significance' significanceLevel=BigDecimalType)?
    '}';

ExperimentalObject returns ExperimentalObject:
    {ExperimentalObject}
    name=ID
    '{'
    'description' description=STRING
    ('value' value=STRING)?
    ('group' objectGroup=[ObjectGroup|ID])?
    ('parameters' '{' parameters+=Parameter (","
       parameters+=Parameter)* '}')?
    ('files' '{' files+=File ("," files+=File)* '}')?
    '}'
;
Abstract_Impl returns Abstract:
    {Abstract};

Author returns Author:
    {Author}
    name=ID
    '{'
    ('fullName' fullName=STRING)?
    ('institution' institution=STRING)?
    ('email' email=STRING)?

    '}';
```

```

Keyword returns Keyword:
    {Keyword}
    description=STRING
    ;

Threat returns Threat:
    {Threat}
    name=ID
    '{'
    ('description' description=STRING)?
    ('type' type=ThreatType)?
    ('CA' CA=STRING)?
    '}';

Goal_Impl returns Goal:
    {Goal}
    name=ID;

ResearchQuestion returns ResearchQuestion:
    {ResearchQuestion}
    name=ID
    '{'
    ('description' description=STRING)?
    ('goal' goal=[Goal|ID])?
    '}';

ResearchHypothesis returns ResearchHypothesis:
    {ResearchHypothesis}
    name=ID
    '{'
    formula=ResearchHypothesisFormula
    ('description' description=STRING)?
    ('goal' goal=[Goal|ID])?
    '}';

ResearchHypothesisFormula returns ResearchHypothesisFormula:
    {ResearchHypothesisFormula}
    depVariable=[DependentVariable|ID] treatment1=[Treatment|ID]
       operator=OperatorType treatment2=[Treatment|ID];

OperatorType:
    typeName=('<' | '=' |'!=' | '>');

DependentVariable returns DependentVariable:
    {DependentVariable}
    name=ID
    '{'
    'description' description=STRING
    ('scaleType' scaleType=ScaleType)?
    ('unit' unit=STRING)?
    ('range' '{' range+=Range ("," range+=Range)* '}')?
    ('instrument' instrument=[Instrument|ID])?
    '}';
Instrument returns Instrument:
    {Instrument}
    name=ID
    '{'
    'command' command=STRING
    'valueExpression' valueExpression=STRING
    ('conversionFactor' conversionFactor=BigDecimalType)?
    '}'
;
Factor returns Factor:
    {Factor}
    name=ID
    '{'
    'description' description=STRING
    ('scaleType' scaleType=ScaleType)?
    '}';

ContextVariable returns Context:
    {Context}
    name=ID
    '{'
    ('description' description=STRING)?
    ('scaleType' scaleType=ScaleType)?
    ('range' '{' range+=Range ("," range+=Range)* '}')?
    '}';

enum DesignType returns DesignType:

    FACTORIAL='FACTORIAL' |CRD='CRD' | RCBD='RCBD' | LS='LS' |
        OTHER='OTHER';

enum ScaleType returns ScaleType:
    Absolute='Absolute' | Logarithmic='Logarithmic' | Nominal='Nominal';

Range returns Range:
    {Range}
    name=ID;

Treatment returns Treatment:
    name=ID
    'description' description=STRING
    'factor' factor=[Factor|ID]
    ('parameters' '{' parameters+=Parameter (","
       parameters+=Parameter)* '}')?
    ('files' '{' files+=File ("," files+=File)* '}')?
    'execution' execution=[Execution|ID];

File returns File:
    {File}
    '{'
    'name' name=STRING
    'source' source=STRING
    ('dest' dest=STRING)?
    ('checksum' checksum=STRING)?
    '}'
;
Parameter returns Parameter:
    {Parameter}
    name=ID
    (value=STRING)?;

ObjectGroup returns ObjectGroup:
    {ObjectGroup}
    name=ID;

SimpleAbstract returns SimpleAbstract:
    {SimpleAbstract}
    (description=STRING)
    ;

StructuredAbstract returns StructuredAbstract:
    {StructuredAbstract}
    '{'
    ('context' context=STRING)?
    ('objective' objective=STRING)?
    ('method' method=STRING)?
    ('results' results=STRING)?
    ('conclusion' conclusion=STRING)?
    '}';

enum ThreatType returns ThreatType:
    iv='iv' | ev='ev' | c='c' | r='r' | cl='cl';

SimpleGoal returns SimpleGoal:
    {SimpleGoal}
    name=ID
    description=STRING
    ;

StructuredGoal returns StructuredGoal:
    {StructuredGoal}
    name=ID
    '{'
    ('object' object=STRING)?
    ('technique' technique=STRING)?
    ('quality' quality=STRING)?
    ('ptView' ptView=STRING)?
    ('contextOf' contextOf=STRING)?
    '}';
```

Appendix B

DSL Validators

In this appendix, we present the validators of the DSL (Listing B.1) created using Xtend and integrated with Xtext framework.

Listing B.1: DSL Validators

```
package br.unb.autoexp.validation

import br.unb.autoexp.autoExp.AutoExpPackage
import br.unb.autoexp.autoExp.DependentVariable
import br.unb.autoexp.autoExp.Execution
import br.unb.autoexp.autoExp.Experiment
import br.unb.autoexp.autoExp.Factor
import br.unb.autoexp.autoExp.ResearchHypothesis
import br.unb.autoexp.autoExp.ResearchHypothesisFormula
import br.unb.autoexp.autoExp.Treatment
import br.unb.autoexp.generator.ExperimentalDesignGenerator
import javax.inject.Inject
import org.eclipse.xtext.validation.Check

import static extension java.lang.String.*


class AutoExpValidator extends AbstractAutoExpValidator {
  public static val ISSUE_CODE_PREFIX = "br.unb.autoexp.";
  public static val HIERARCHY_CYCLE = ISSUE_CODE_PREFIX +
     "HierarchyCycle";
  public static val INVALID_ENTITY_NAME = ISSUE_CODE_PREFIX +
     "InvalidEntityName";
  public static val INVALID_ATTRIBUTE_NAME = ISSUE_CODE_PREFIX +
     "InvalidAttributeName";
```

```
public static val INVALID_PARAMETER_PLACEHOLDER = ISSUE_CODE_PREFIX +
   "InvalidParameterPlaceholder";
public static val SAME_TREATMENT_COMPARISON = ISSUE_CODE_PREFIX +
   "SameTreatmentComparison";
public static val TREATMENT_FROM_DISTINCT_FACTORS = ISSUE_CODE_PREFIX
   + "InvalidTreatment";
public static val SAME_FORMULA = ISSUE_CODE_PREFIX + "SameFormula";
public static val DEPENDENT_VARIABLE_NEVER_USED = ISSUE_CODE_PREFIX +
   "DependentVariableNeverUsed"
public static val FACTOR_NEVER_USED = ISSUE_CODE_PREFIX +
   "FactorNeverUsed"
public static val TREATMENT_NEVER_USED = ISSUE_CODE_PREFIX +
   "TreatmentNeverUsed"
public static val EXECUTION_NEVER_USED = ISSUE_CODE_PREFIX +
   "ExecutionNeverUsed"
public static val INVALID_PARAMETER = ISSUE_CODE_PREFIX +
   "InvalidParameter"
public static val UNREGISTERED_DESIGN = ISSUE_CODE_PREFIX +
   "UnregisteredDesign"

@Inject extension ExperimentalDesignGenerator

@Check
def checkRepeatedHypothesis(ResearchHypothesis hypothesis) {
  val experiment = hypothesis.eContainer as Experiment
  experiment.researchHypotheses.forEach [ hyp |
    if (!hypothesis.name.equals(hyp.name) &&
      hypothesis.formula.depVariable
    .equals(hyp.formula.depVariable) &&
      hypothesis.formula.treatment1
        .equals(hyp.formula.treatment1)  &&
      hypothesis.formula.operator.typeName
        .equals(hyp.formula.operator.typeName) &&
      hypothesis.formula.treatment2
        .equals(hyp.formula.treatment2)) {
        warning("Hyphoteses '%s' and '%s' have the same
           formula".format(hypothesis.name, hyp.name),
        AutoExpPackage.eINSTANCE.researchHypothesis_Formula,
           AutoExpValidator.SAME_FORMULA, experiment.name)
    }
  ]
}
@Check
def checkSameTreatmentComparison(ResearchHypothesisFormula
   hyphotesisFormula) {
```

```
  if (hyphotesisFormula.treatment1
     .equals(hyphotesisFormula.treatment2))
     error("Comparison must be done between distinct treatments",
       AutoExpPackage.eINSTANCE
         .researchHypothesisFormula_Treatment2,
         SAME_TREATMENT_COMPARISON, hyphotesisFormula.treatment2.name)
}
@Check
def checkTreatmentsFromDistinctFactors(ResearchHypothesisFormula
   hyphotesisFormula) {
   if (!hyphotesisFormula.treatment1.factor
     .equals(hyphotesisFormula.treatment2.factor))
      error("Treatments '%s' and '%s' do not belong to the same
         factor".format(hyphotesisFormula.treatment1.name,
        hyphotesisFormula.treatment2.name),
        AutoExpPackage.eINSTANCE
          .researchHypothesisFormula_Treatment2,
    TREATMENT_FROM_DISTINCT_FACTORS,
       hyphotesisFormula.treatment2.name)
}
@Check
def checkDependentVariableNeverUsed(DependentVariable variable) {
  val experiment = variable.eContainer as Experiment
  if (!experiment.researchHypotheses.map[formula.depVariable]
    .contains(variable)) {
    warning("Dependent variable '%s' is never
       used".format(variable.name),
      AutoExpPackage.eINSTANCE.dependentVariable_Name,
         AutoExpValidator.DEPENDENT_VARIABLE_NEVER_USED,
         variable.name)
  }
}
@Check
def checkInvalidParameter(Execution execution) {
  val experiment = execution.eContainer as Experiment
  experiment.designExecutions.filter[execution.name.equals(name)]
    .forEach [ exec |
    exec.invalidParameters.forEach [ parameter,attribute |
        val att=switch attribute{
      case "cmd": AutoExpPackage.eINSTANCE.execution_Cmd
      case "result": AutoExpPackage.eINSTANCE.execution_Result
    }
      error("Parameter '%s' cannot be resolved".format(parameter),
         att, AutoExpValidator.INVALID_PARAMETER, parameter)
    ]
  ]
}
@Check
def checkFactorNeverUsed(Factor factor) {
  val experiment = factor.eContainer as Experiment
  if (!experiment.researchHypotheses
    .map[formula.treatment1.factor].contains(factor) &&
    !experiment.researchHypotheses
    .map[formula.treatment2.factor].contains(factor)) {
      warning("Factor '%s' is never used".format(factor.name),
         AutoExpPackage.eINSTANCE.factor_Name,
        AutoExpValidator.FACTOR_NEVER_USED, factor.name)
  }
}
@Check
def checkTreatmentNeverUsed(Treatment treatment) {
  val experiment = treatment.eContainer as Experiment
  if (!experiment.researchHypotheses.map[formula.treatment1]
    .contains(treatment) &&
       !experiment.researchHypotheses.map[formula.treatment2]
    .contains(treatment)) {
    warning("Treatment '%s' is never used".format(treatment.name),
       AutoExpPackage.eINSTANCE.treatment_Name,
  AutoExpValidator.TREATMENT_NEVER_USED, treatment.name)
  }
}
@Check
def checkExecutionNeverUsed(Execution execution) {
  val experiment = execution.eContainer as Experiment
    if (!experiment.researchHypotheses
      .map[formula.treatment1.execution].contains(execution) &&
      !experiment.researchHypotheses
      .map[formula.treatment2.execution].contains(execution)) {
    warning("Execution '%s' is never used".format(execution.name),
       AutoExpPackage.eINSTANCE.execution_Name,
      AutoExpValidator.EXECUTION_NEVER_USED, execution.name)
    }
}

}
```

Appendix C

Generators

In this appendix, we present the execution script generator (Listing C.1) and the analysis script generator (Listing C.2) for the DSL.

Listing C.1: Execution Script Generator

```
package br.unb.autoexp.generator.dohko

import br.unb.autoexp.autoExp.Experiment
import br.unb.autoexp.generator.ExperimentalDesignGenerator
import javax.inject.Inject

class DohkoGenerator {
@Inject extension ExperimentalDesignGenerator
  def compileDohko(Experiment experiment) {
    '''
    ---
    name: "«experiment.name»"
    description: "«IF experiment.description!==null» «
       experiment.description» «ENDIF»"
    user:
      username: "«experiment.infrastructure.user.username»"
      «IF !experiment.infrastructure.user.keys.isNullOrEmpty»
      keys:
      «ENDIF»
        «FOR key:experiment.infrastructure.user.keys»
        - name: "«key.name»"
          «IF key.privateKey!==null»
          private-key-material: "«key.privateKey»"
          «ENDIF»
          «IF key.privateKey!==null»
          public-key-material: "«key.publicKey»"
          «ENDIF»
          «IF key.privateKey!==null»
          fingerprint: "«key.fingerprint»"
          «ENDIF»
        «ENDFOR»
    «IF experiment.infrastructure.requirements!==null»
        requirements:
          cpu: «experiment.infrastructure.requirements.cpu»
          memory: «experiment.infrastructure.requirements.memory»
          platform: "«experiment.infrastructure
                      .requirements.platform.typeName»"
          cost: «experiment.infrastructure.requirements.cost»
          number-of-instances-per-cloud: «experiment.infrastructure
                       .requirements.instancesPerCloud»
    «ENDIF»
    «IF experiment.infrastructure.preconditions!==null»
        preconditions:
          packages:
          «FOR pack:experiment.infrastructure
                    .preconditions.packages»
          - «pack»
          «ENDFOR»
    «ENDIF»
    «IF !experiment.infrastructure.clouds.isNullOrEmpty»
        clouds:
          «FOR cloud:experiment.infrastructure.clouds»
            - name: "«cloud.name»"
              «IF cloud.provider!==null»
              provider:
                name: "«cloud.provider.name»"
                «IF cloud.provider.maxResourcePerType>0»
                max-resource-per-type: «
                   cloud.provider.maxResourcePerType»
                «ENDIF»
                «IF cloud.provider.description!==null»
                description: "«cloud.provider.description»"
                «ENDIF»
                «IF cloud.provider.serviceClass!==null»
                service-class: "«cloud.provider.serviceClass»"
                «ENDIF»
              «ENDIF»
              «IF cloud.accessKey!==null»
              access-key:
                access-key: "«cloud.accessKey.accessKey»"
                secret-key: "«cloud.accessKey.secretKey»"
```

```
              «ENDIF»
              «IF !cloud.regions.isNullOrEmpty»
                regions:
              «ENDIF»
              «FOR region:cloud.regions»
                - name: "«region.name»"
                  «IF region.endpoint!==null»
                  endpoint: "«region.endpoint»"
                  «ENDIF»
                  «IF region.status!==null»
                  status: «region.status.typeName»
                  «ENDIF»
                  «IF region.city!==null»
                  city: "«region.city»"
                  «ENDIF»
                  «IF region.geographicRegion!=0»
                  geographic-region: «region.geographicRegion»
                  «ENDIF»
                  «IF !region.zones.isNullOrEmpty»
                  zone:
                  «ENDIF»
                  «FOR zone:region.zones»
                  - name: "«zone.name»"
                    «IF zone.status!==null»
                    status: "«zone.status»"
                    «ENDIF»
                  «ENDFOR»
              «ENDFOR»
              «IF !cloud.instanceTypes.isNullOrEmpty»
              instance-types:
              «ENDIF»
                «FOR instance:cloud.instanceTypes»
                - name: "«instance.name»"
                  «IF instance.numberOfInstances>0»
                  number-of-instances: «instance.numberOfInstances»
                  «ENDIF»
                «ENDFOR»
          «ENDFOR»
    «ENDIF»

    «IF !experiment.designExecutions.isNullOrEmpty»
        blocks:
        «FOR execution:experiment.designExecutions»
          - repeat: «experiment.experimentalDesign.runs»
            applications:
              - name: "«execution.taskName»"
                command-line: "«execution.cmd»"
                «IF execution.timeout!==null»
                timeout: «execution.timeout»
                «ENDIF»
                «IF execution.preconditions!==null»
                preconditions:
                  packages:
                  «FOR pack:execution.preconditions.packages»
                  - «pack»
                  «ENDFOR»
                «ENDIF»
                «IF!execution.files.isNullOrEmpty»
                files:
                «ENDIF»
                «FOR file:execution.files»
                - name: "«file.name»"
                  path: "«file.path»"
                  generated: «IF file.generated»"Y"«ELSE»"N"«ENDIF»
               «ENDFOR»
        «ENDFOR»
    «ENDIF»
    «IF experiment.infrastructure.onFinish!==null»
        on-finish: "«experiment.infrastructure.onFinish.typeName»"
    «ENDIF»
    '''

  }

}
```

Listing C.2: Analysis Script Generator

```
package br.unb.autoexp.generator.rscript

import br.unb.autoexp.autoExp.DependentVariable
import br.unb.autoexp.autoExp.Experiment
import br.unb.autoexp.autoExp.ExperimentalObject
import br.unb.autoexp.autoExp.ResearchHypothesis
import br.unb.autoexp.autoExp.ScaleType
import br.unb.autoexp.autoExp.SimpleGoal
import br.unb.autoexp.autoExp.Treatment
import br.unb.autoexp.autoExp.impl.SimpleAbstractImpl
import br.unb.autoexp.autoExp.impl.SimpleGoalImpl
import br.unb.autoexp.autoExp.impl.StructuredAbstractImpl
import br.unb.autoexp.autoExp.impl.StructuredGoalImpl
import br.unb.autoexp.generator.ExperimentalDesignGenerator
import java.util.List
import javax.inject.Inject

class RScriptGenerator {
@Inject extension ExperimentalDesignGenerator
  def compileRScript(Experiment experiment) {
        '''
        \documentclass{article}
        \usepackage{authblk}
        \usepackage{float}
        \usepackage{multirow}
        \usepackage[utf8]{inputenc}
        \begin{document}
        «experiment.generateTitle»
        «experiment.generateAuthor»
        \maketitle
        «experiment.generateAbstract»
        «experiment.generateKeywords»
        <<setup, include=FALSE, echo=FALSE, warning=FALSE ,
           message=FALSE >≥
        library(reproducer) # R package incl. software engineering data
           sets
        library(ggplot2) # R package to create high-quality graphics
        library(jsonlite)

        alpha = «IF experiment.analysis?.significanceLevel !== null» «
           experiment.analysis.significanceLevel»«ELSE»0.05«ENDIF»

        json_data = fromJSON("data.json")

        «FOR i:1..experiment.experimentalObjects.size»
      json_data$objectOrder[json_data$object ==
         '«experiment.experimentalObjects.get(i-1).name»'] = «i»
        «ENDFOR»

        «FOR treatment:experiment.treatmentsInUse»
      json_data$treatmentDescription[json_data$treatment ==
         '«treatment.name»'] = '«treatment.description»'
        «ENDFOR»
        «FOR object:experiment.experimentalObjects»
```

```
    json_data$objectLabel[json_data$object == '«object.name»'] = '«IF
       object.value === null» «
       object.description»«ELSE»«object.value»«ENDIF»'
      «ENDFOR»

      expectedRuns = «experiment.experimentalDesign.runs»
      «FOR variable: (experiment.researchHypotheses as
         List<ResearchHypothesis>).map[
         formula.depVariable].removeDuplicates»
    json_data$«variable.name.convert »[json_data$executionStatus !=
       'FINISHED'] = NA
    «FOR treatment:experiment.treatmentsInUse»
  «FOR object:treatment.experimentalObjects»
      if (length(json_data$«variable.name.convert»[
         json_data$treatment == '«treatment.name»' & json_data$object
         == '«object.name»' & !is.na(
         json_data$«variable.name.convert»)]) != expectedRuns){
          json_data$«variable.name.convert »[json_data$treatment ==
             '«treatment.name»' & json_data$object ==
             '«object.name»']=NA
      }
  «ENDFOR»
    «ENDFOR»
      «ENDFOR»

      json_data$treatment = as.factor(json_data$treatment)
      json_data$treatmentDescription =
         as.factor(json_data$treatmentDescription)
      json_data$object = as.factor(json_data$object)
      «IF experiment.objectsScaleType.equals(ScaleType.NOMINAL)»
    json_data$objectLabel = as.factor(json_data$objectLabel)
      «ELSE»
    json_data$objectLabel = as.numeric(json_data$objectLabel)
      «ENDIF»
      data_summary <- function(data, varname, groupnames){
  require(plyr)
  summary_func <- function(x, col){
    c(mean = mean(x[[col]], na.rm=TRUE),
      sd = sd(x[[col]], na.rm=TRUE))
  }
  data_sum<-ddply(data, groupnames, .fun=summary_func,
            varname)
  data_sum <- rename(data_sum, c("mean" = varname))
       return(data_sum)
      }
      breaks_continuous <- function(data, steps){
  diff<-max(data)-min(data)
  step_size<-diff/steps
  step<-min(data)
  breaks<-c(step)
  for (i in 1:steps){
    step<-step+step_size
    breaks<-c(breaks,step)
  }
  return(breaks)
      }
      breaks_log <- function(data, steps){
  diff<-max(data)/min(data)
  base<-diff^(1/steps)
  exp<-log(min(data),base)
  breaks<-c(round(base^exp))
  for (i in 1:steps){
    exp<-exp+1
    breaks<-c(breaks,round(base^exp))
  }
  return(breaks)
      }
      @
      \section{Description}
      «experiment.description»
      «experiment.generateGoals»
      «experiment.generateQuestions»

      \section{Overview}
      «experiment.generateOverview»

      \subsection{Objects Overview}
      «FOR object:experiment.objectsInUse»
    \subsubsection{Overview for «object.description»}
    «experiment.generateObjectOverview(object)»
      «ENDFOR»

      \section{Research Hypotheses}
      «FOR hypothesis:experiment.researchHypotheses»

      \subsection{«hypothesis.name»: «hypothesis.description»}
      «hypothesis.generate»

      «ENDFOR»

```

```
        \section{Result Summary}
        \subsection{Research Hypotheses}

        «experiment.generateResultsSummary»

        «experiment.generateResultsFile»

        «generateSessionInformation»

        \end{document}
        '''
    }

    def String generateResultsSummary(Experiment experiment)
        '''
        «FOR hypothesis:experiment.researchHypotheses»
      «hypothesis.generateSummary»
        «ENDFOR»

        '''

    def String generateSessionInformation() {
        '''
      \clearpage
      \appendix
      \section{Session Information}
      <<echo=FALSE, warning=FALSE , message=FALSE >≥
      sessionInfo()
      @
        '''
    }

    def String generate(ResearchHypothesis hypothesis){
        '''

         «hypothesis.initializeResults»

         «hypothesis.generateOverview»

        «FOR obj:hypothesis.objects»

        \subsubsection{«hypothesis.name».«
           hypothesis.objects.indexOf(obj) + 1»: Object «
           obj.description»}
        «hypothesis.generate(obj)»

        «ENDFOR»


        «hypothesis.generateSummary»

        '''
    }
    def String generateOverview(ResearchHypothesis hypothesis){
        val experiment=hypothesis.eContainer as Experiment
        '''
      <<overview_«hypothesis.name», include=TRUE, echo=FALSE,
         warning=FALSE , message=FALSE >≥
      DF <- data_summary(subset(json_data, «FOR
         object:hypothesis.objectsInUse BEFORE "(" SEPARATOR "|" AFTER
         ")"»object == '«object.name»' «ENDFOR» & (treatment ==
         '«hypothesis.formula.treatment1.name»' | treatment ==
         '«hypothesis.formula.treatment2.name»')), varname =
         "«hypothesis.formula.depVariable.name.convert»", groupnames =
         c("treatmentDescription", "objectLabel", "objectOrder"))
      «generatePlotOverview(experiment, hypothesis.formula.depVariable)»
      @
        '''
    }


    def String generateResultsFile(Experiment experiment)
        '''
        <<echo=TRUE, echo=FALSE, warning=FALSE , message=FALSE >≥
        experimentResults = list(«FOR hypothesis:
           experiment.researchHypotheses» «hypothesis.name»_result«IF
           !hypothesis.name.equals(
           experiment.researchHypotheses.last.name)», «ENDIF»«ENDFOR»)
        write(toJSON(experimentResults, pretty = TRUE, auto_unbox =
           TRUE), "experimentResults.json")

        @
        '''

    def String generateSummary(ResearchHypothesis hypothesis)
        '''
        <<echo=FALSE, echo=FALSE, warning=FALSE , message=FALSE >≥
        «hypothesis.name»_result = list(hypothesis =
           "«hypothesis.name»", results =
           c(result_«hypothesis.name»_less /
           result_«hypothesis.name»_objects,
           result_«hypothesis.name»_greater /
           result_«hypothesis.name»_objects, result_«hypothesis.name»_«
           hypothesis.formula.treatment1.name» /
           result_«hypothesis.name»_objects, result_«hypothesis.name»_«
           hypothesis.formula.treatment2.name» /
           result_«hypothesis.name»_objects,
           result_«hypothesis.name»_none /
           result_«hypothesis.name»_objects,
           result_«hypothesis.name»_inconclusive /
           result_«hypothesis.name»_objects), objectResults = list(«FOR
           object:hypothesis.objects» list(object = '«object.name»',
           result = result_object_«hypothesis.name»_«object.name»)«IF
           !object.name.equals(hypothesis.objects.last.name)», «
           ENDIF»«ENDFOR» ))
        @

        \subsubsection{«hypothesis.name» Results: «
           hypothesis.formula.depVariable.description» «
           hypothesis.formula.treatment1.description» «
           hypothesis.formula.operator.typeName» «
           hypothesis.formula.treatment2.description»}


        \begin{table}[H]
        \centering
        \caption{«hypothesis.name» Results per Object}
        \begin{tabular}{ll}
        «FOR object:hypothesis.objects»
        \textbf{«object.description»} &
           \Sexpr{result_«hypothesis.name»_«object.name»} \\
        «ENDFOR»
        \end{tabular}
        \end{table}

        \begin{table}[H]
        \centering
        \caption{«hypothesis.name» Results Summary}
        \begin{tabular}{ll}
        \textbf{«hypothesis.formula.treatment1.description» \textless{}
           «hypothesis.formula.treatment2.description»:}& \Sexpr{100 *
           result_«hypothesis.name»_less /
           result_«hypothesis.name»_objects}\% \\
        \textbf{«hypothesis.formula.treatment1.description»
           \textgreater{} «
```

```
          hypothesis.formula.treatment2.description»:}& \Sexpr{100 *
          result_«hypothesis.name»_greater /
          result_«hypothesis.name»_objects}\%\\
        \textbf{«hypothesis.formula.treatment1.description»:} &
          \Sexpr{100 * result_«hypothesis.name»_«
          hypothesis.formula.treatment1.name» /
          result_«hypothesis.name»_objects}\%\\
        \textbf{«hypothesis.formula.treatment2.description»:} &
          \Sexpr{100 * result_«hypothesis.name»_«
          hypothesis.formula.treatment2.name» /
          result_«hypothesis.name»_objects}\%\\
        \textbf{None:}& \Sexpr{100 * result_«hypothesis.name»_none /
          result_«hypothesis.name»_objects}\%\\
        \textbf{Inconclusive:}& \Sexpr{100 *
          result_«hypothesis.name»_inconclusive /
          result_«hypothesis.name»_objects}\%
        \end{tabular}
        \end{table}
        '''

    def String generate(ResearchHypothesis hypothesis,
       ExperimentalObject object){
        '''

    «hypothesis.generateTreatmentsData(object)»

    \textbf{Comparison}

    <<«hypothesis.name»_«object.name», include=TRUE, echo=FALSE,
       warning=FALSE, message=FALSE >≥
    «hypothesis.generateBoxplot(object)»
    if( length( «hypothesis.formula.depVariable.name.convert»_«
       hypothesis.formula.treatment1.name»_« object.name») ==
       expectedRuns & length( «
       hypothesis.formula.depVariable.name.convert»_«
       hypothesis.formula.treatment2.name»_«object.name») ==
       expectedRuns){
        «hypothesis.generateTests(object)»
    }
         if( length( «hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment1.name»_« object.name») ==
            expectedRuns & length( «
            hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment2.name»_« object.name») ==
            expectedRuns){
```

```
        «hypothesis.generateComparison(object)»
         }
         if ( length( «hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment1.name»_« object.name») !=
            expectedRuns & length( «
            hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment2.name»_« object.name») !=
            expectedRuns){
        result_object_«hypothesis.name»_«object.name» = 4
       result_«hypothesis.name»_«object.name» = "None"
       result_«hypothesis.name»_none = result_«hypothesis.name»_none + 1
         }
         if ( length( «hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment1.name»_«object.name») ==
            expectedRuns & length( «
            hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment2.name»_«object.name») !=
            expectedRuns){
       result_object_«hypothesis.name»_«object.name» = 2
       result_«hypothesis.name»_«object.name» =
          "«hypothesis.formula.treatment1.description»"
       result_«hypothesis.name»_« hypothesis.formula.treatment1.name» =
          result_« hypothesis.name»_«
          hypothesis.formula.treatment1.name» + 1
         }
         if ( length( «hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment1.name»_« object.name») !=
            expectedRuns & length( «
            hypothesis.formula.depVariable.name.convert»_«
            hypothesis.formula.treatment2.name»_«object.name») ==
            expectedRuns){
       result_object_«hypothesis.name»_«object.name» = 3
       result_«hypothesis.name»_«object.name» =
          "«hypothesis.formula.treatment2.description»"
       result_« hypothesis.name»_« hypothesis.formula.treatment2.name»
          = result_« hypothesis.name»_«
          hypothesis.formula.treatment2.name» + 1
         }
         @
        '''
        }

    def String initializeResults(ResearchHypothesis hypothesis)
        '''
        <<«hypothesis.name», include=TRUE, echo=FALSE, warning=FALSE ,
           message=FALSE >≥

        result_«hypothesis.name»_objects=«hypothesis.objects.size»
        result_«hypothesis.name»_less=0
        result_«hypothesis.name»_greater=0
        result_«hypothesis.name»_« hypothesis.formula.treatment1.name»
           = 0
        result_«hypothesis.name»_« hypothesis.formula.treatment2.name»
           = 0
        result_«hypothesis.name»_none = 0
        result_«hypothesis.name»_inconclusive = 0
        @
        '''
    def String generateComparison(ResearchHypothesis hypothesis,
        ExperimentalObject  object)
      '''
      print("")
      print("Means comparison")
      print( paste( "Mean «hypothesis.formula.depVariable.description»
         for «hypothesis.formula.treatment1.description»: ", mean(
         subset( json_data, treatment ==
         '«hypothesis.formula.treatment1.name»' & object ==
         '«object.name»')$«
         hypothesis.formula.depVariable.name.convert»)))
      print( paste( "Mean «hypothesis.formula.depVariable.description»
         for «hypothesis.formula.treatment2.description»: ", mean(
         subset( json_data, treatment ==
         '«hypothesis.formula.treatment2.name»' & object ==
         '«object.name»')$«
         hypothesis.formula.depVariable.name.convert»)))
      print( paste( "Absolute difference: ", abs( mean( subset(
         json_data, treatment == '«hypothesis.formula.treatment1.name»'
         & object == '«object.name»')$«
         hypothesis.formula.depVariable.name.convert») - mean( subset(
         json_data, treatment == '«hypothesis.formula.treatment2.name»'
         & object == '«object.name»')$«
         hypothesis.formula.depVariable.name.convert»))))
      if (result_« hypothesis.name»_«object.name»_tTest |
         result_«hypothesis.name»_« object.name»_wTest){
          if( mean( subset( json_data, treatment ==
             '«hypothesis.formula.treatment1.name»' & object ==
             '«object.name»')$«
             hypothesis.formula.depVariable.name.convert») > mean(
             subset( json_data, treatment ==
             '«hypothesis.formula.treatment2.name»' & object ==
             '«object.name»')$«
             hypothesis.formula.depVariable.name.convert»)){
            result_«hypothesis.name»_«object.name» =
               "«hypothesis.formula.treatment1.description»
               \\textgreater{} «
               hypothesis.formula.treatment2.description»"
            result_object_«hypothesis.name»_«object.name» = 1
            result_«hypothesis.name»_greater =
               result_«hypothesis.name»_greater + 1
          }else {
            result_«hypothesis.name»_«object.name» =
               "«hypothesis.formula.treatment1.description»
               \\textless{} «hypothesis.formula.treatment2.description»"
            result_object_«hypothesis.name»_«object.name» = 0
            result_«hypothesis.name»_less =
               result_«hypothesis.name»_less + 1
          }

      }else{
          result_object_«hypothesis.name»_«object.name» = 5
          result_«hypothesis.name»_«object.name» = "Inconclusive"
          result_«hypothesis.name»_inconclusive =
             result_«hypothesis.name»_inconclusive + 1
      }

      if( mean( subset( json_data, treatment ==
         '«hypothesis.formula.treatment1.name»' & object ==
         '«object.name»')$«
         hypothesis.formula.depVariable.name.convert») > mean( subset(
         json_data, treatment == '«hypothesis.formula.treatment2.name»'
         & object == '«object.name»')$«
         hypothesis.formula.depVariable.name.convert» )){
          cat( paste( "«hypothesis.formula.depVariable.description» for
             «hypothesis.formula.treatment1.description» is ", 100 * (
             abs( mean( subset( json_data, treatment == '«
             hypothesis.formula.treatment2.name»' & object ==
             '«object.name»')$«
             hypothesis.formula.depVariable.name.convert») - mean(
             subset( json_data, treatment == '«
             hypothesis.formula.treatment1.name»' & object ==
             '«object.name»')$«
             hypothesis.formula.depVariable.name.convert» )) / mean(
             subset( json_data, treatment == '«
             hypothesis.formula.treatment2.name»' & object ==
             '«object.name»')$«
             hypothesis.formula.depVariable.name.convert» )), "%
             greater than \n «
             hypothesis.formula.depVariable.description» for «
             hypothesis.formula.treatment2.description»" ))
     }else{
         cat( paste( "«hypothesis.formula.depVariable.description» for
             «hypothesis.formula.treatment2.description» is ", 100 * (
             abs( mean( subset( json_data, treatment == '«
             hypothesis.formula.treatment2.name»' & object ==
             '«object.name»')$«
             hypothesis.formula.depVariable.name.convert») - mean(
             subset( json_data, treatment ==
             '«hypothesis.formula.treatment1.name»' & object ==
             '«object.name»' )$«
             hypothesis.formula.depVariable.name.convert» )) / mean(
             subset( json_data, treatment == '«
             hypothesis.formula.treatment1.name»' & object ==
             '«object.name»')$«
             hypothesis.formula.depVariable.name.convert» )), "%
             greater than \n«
             hypothesis.formula.depVariable.description» for «
             hypothesis.formula.treatment1.description»" ))
     }
     '''

   def String generateNonParametricTest(ResearchHypothesis hypothesis,
       ExperimentalObject  object)
       '''
     result_«hypothesis.name»_«object.name»_wTest = FALSE
     wTest = wilcox.test( «hypothesis.formula.depVariable.name.convert
         »¬treatment, data = subset( json_data, (treatment ==
         '«hypothesis.formula.treatment1.name»' | treatment == '«
         hypothesis.formula.treatment2.name»') & object ==
         '«object.name»'))
     print(wTest)
     if(wTest$p.value > alpha){
         print( paste( "Wilcoxon-Mann-Whitney test: Null Hypothesis
             not rejected. P-value:", wTest$p.value, sep = " "))
          result_«hypothesis.name»_« object.name»_wTest = FALSE
     }else{
          print( paste( "Wilcoxon-Mann-Whitney test: Null Hypothesis
             rejected. P-value:", wTest$p.value, sep = " "))
          result_« hypothesis.name»_«object.name»_wTest = TRUE
     }
```

```
        '''

    def String generateTests(ResearchHypothesis hypothesis,
       ExperimentalObject  object)
      '''
        result_«hypothesis.name»_« object.name»_tTest = FALSE
        result_«hypothesis.name»_« object.name»_wTest = FALSE

        if( shap_«hypothesis.formula.treatment1.name»_«
            object.name»$p.value > alpha &
            shap_«hypothesis.formula.treatment2.name»_«
            object.name»$p.value > alpha){
    print("Fisher's F-test to verify the homoskedasticity (homogeneity
       of variances)")

    fTest = var.test( subset( json_data, treatment ==
       '«hypothesis.formula.treatment1.name»' & object ==
       '«object.name»')$« hypothesis.formula.depVariable.name.convert »
       , subset(json_data,treatment == '«
       hypothesis.formula.treatment2.name»' & object ==
       '«object.name»')$« hypothesis.formula.depVariable.name.convert»)
    print(fTest)

    print( paste( "Homogeneity of variances: ", fTest$p.value > alpha,
       ". P-value: ", fTest$p.value, sep = ""))

    print("Assuming that the two samples are taken from populations
       that follow a Gaussian distribution (if we cannot assume that,
       we must solve this problem using the non-parametric test called
       Wilcoxon-Mann-Whitney test)")
    tTest = t.test( subset( json_data, treatment ==
       '«hypothesis.formula.treatment1.name»' & object ==
       '«object.name»')$« hypothesis.formula.depVariable.name.convert »
       , subset( json_data, treatment ==
       '«hypothesis.formula.treatment2.name»' & object ==
       '«object.name»')$« hypothesis.formula.depVariable.name.convert»,
       var.equal = fTest$p.value > alpha, paired = FALSE)
    print(tTest)
    if(tTest$p.value > alpha){
      print(paste("T-test: Null Hypothesis not rejected. P-value:",
         tTest$p.value, sep = " "))

    }else{
      print(paste("T-test: Null Hypothesis rejected. P-value:",
         tTest$p.value, sep = " "))
      result_«hypothesis.name»_«object.name»_tTest = TRUE
    }
        }else{
    wTest = wilcox.test( «hypothesis.formula.depVariable.name.convert »¬
       treatment, data = subset( json_data, (treatment ==
       '«hypothesis.formula.treatment1.name»' | treatment ==
       '«hypothesis.formula.treatment2.name»') & object ==
       '«object.name»'))
    print(wTest)
    if(wTest$p.value > alpha){
      print( paste( "Wilcoxon-Mann-Whitney test: Null Hypothesis not
         rejected. P-value:", wTest$p.value, sep = " "))
      result_«hypothesis.name»_« object.name»_wTest = FALSE
    }else{
      print( paste( "Wilcoxon-Mann-Whitney test: Null Hypothesis
         rejected. P-value:", wTest$p.value, sep = " "))
      result_«hypothesis.name»_« object.name»_wTest = TRUE
    }
        }
       '''
  def String generateBoxplot(ResearchHypothesis hypothesis,
     ExperimentalObject  object)
  '''
      DF=subset(json_data,(treatment ==
         '«hypothesis.formula.treatment1.name»' | treatment ==
         '«hypothesis.formula.treatment2.name»') & object ==
         '«object.name»')
      DF$treatmentDescription = ordered(DF$treatmentDescription, levels
         = levels(DF$treatmentDescription)[order( as.numeric( by(
         DF$«hypothesis.formula.depVariable.name.convert»,
         DF$treatmentDescription, mean)))])
      boxplot_«hypothesis.name»_«object.name» = ggplot(DF, aes(x
         =treatmentDescription , y = «
         hypothesis.formula.depVariable.name.convert»)) +
    geom_boxplot(fill = "#4271AE", colour = "#1F3552",alpha =
       0.7,outlier.colour = "#1F3552", outlier.shape = 20)+
    theme_bw() +
    scale_x_discrete(name =
       "«hypothesis.formula.treatment1.factor.description»")+
    ggtitle( "«hypothesis.formula.depVariable.description» by «
       hypothesis.formula.treatment1.factor.description» for «
       object.description»") +
    ylab("«hypothesis.formula.depVariable.description» «IF
       hypothesis.formula.depVariable.unit !== null»( «
       hypothesis.formula.depVariable.unit» )«ENDIF»")
      boxplot_«hypothesis.name»_«object.name»
        '''

    def String generateTreatmentsData(ResearchHypothesis hypothesis, ExperimentalObject  object)
        '''
      «FOR treatment:hypothesis.getTreatments»
    \textbf{«hypothesis.formula.depVariable.description» for «treatment.description»}
       <<«hypothesis.name»_«treatment.name»_«object.name», include = TRUE, echo = FALSE, warning = FALSE , message = FALSE >>=
       «hypothesis.formula.depVariable.name.convert»_«treatment.name»_«object.name» = subset( json_data, treatment == '«treatment.name»' & object == '«object.name»' & !is.na( «hypothesis.formula.depVariable.name.convert» ))$«hypothesis.formula.depVariable.name.convert»
    print(paste("Sample size: ", length(«hypothesis.formula.depVariable.name.convert»_«treatment.name»_«object.name»)))
    summary(subset(json_data, treatment == '«treatment.name»' & object == '«object.name»')$«hypothesis.formula.depVariable.name.convert»)

    if( length( «hypothesis.formula.depVariable.name.convert»_«treatment.name»_«object.name») == expectedRuns){
        reproducer::boxplotAndDensityCurveOnHistogram( subset(json_data, treatment == '«treatment.name»' & object == '«object.name»'), "«hypothesis.formula.depVariable.name.convert»", min(subset(json_data, treatment == '«treatment.name»' & object == '«object.name»')$«hypothesis.formula.depVariable.name.convert»), max( subset(json_data, treatment == '«treatment.name»' & object == '«object.name»')$«hypothesis.formula.depVariable.name.convert»))

        shap_«treatment.name»_«object.name» = shapiro.test( subset(json_data, treatment == '«treatment.name»' & object == '«object.name»')$«hypothesis.formula.depVariable.name.convert»)
        print(shap_«treatment.name»_«object.name»)
        if(shap_«treatment.name»_«object.name»$p.value > alpha){
            print( paste( "Shapiro test: Null Hypothesis (normality) not rejected. P-value:", shap_«treatment.name»_«object.name»$p.value, sep = " "))
        }else{
            print( paste( "Shapiro test: Null Hypothesis (normality)
               rejected. P-value:",
               shap_«treatment.name»_«object.name»$p.value, sep = " "))
        }
    }
    @
      «ENDFOR»
        '''

    def String generateTitle(Experiment experiment)
        '''
      \title{«experiment.description»}
        '''

    def String generateAuthor(Experiment experiment)
        '''
      \author{«FOR author:experiment.authors»«author.fullName»«IF
         !author.name.equals(experiment.authors.last.name)», «
         ENDIF»«ENDFOR»}
        '''

    def String generateAbstract(Experiment experiment)
        '''

        «IF experiment.abstract?.class?.equals(SimpleAbstractImpl)»
      \abstract{«(experiment.abstract as
         SimpleAbstractImpl).description»}
        «ELSEIF experiment.abstract?.class?.equals(
           StructuredAbstractImpl)»
      «val abstract = (experiment.abstract as StructuredAbstractImpl)»
      \begin{abstract}
      «IF !abstract.context.isNullOrEmpty»
      \textbf{Context:} «abstract.context»
      «ENDIF»

      «IF !abstract.objective.isNullOrEmpty»
      \textbf{Objective:} «abstract.objective»
      «ENDIF»

      «IF !abstract.method.isNullOrEmpty»
      \textbf{Method:} «abstract.method»
      «ENDIF»

      «IF !abstract.results.isNullOrEmpty»
```

```
  \textbf{Results:} «abstract.results»
  «ENDIF»

  «IF !abstract.conclusion.isNullOrEmpty»
  \textbf{Conclusion:} «abstract.conclusion»
  «ENDIF»
  \end{abstract}

    «ENDIF»

    '''

def String generateKeywords(Experiment experiment)
    '''
    «IF !experiment.keywords.isNullOrEmpty»
  %\keywords{«FOR keyword : experiment.keywords»«keyword.description»«IF !keyword.equals(experiment.keywords.last)», «ENDIF»«ENDFOR»}
    «ENDIF»

    '''
def String generateGoals(Experiment experiment)
    '''
    «IF !experiment.goals.isNullOrEmpty»
    \section{Goals}
    \begin{itemize}
    «FOR goal:experiment.goals»
  «IF goal.class.equals(SimpleGoalImpl)»
\item{«(goal as SimpleGoal).name»: «(goal as SimpleGoal).description»}
  «ELSEIF goal.class.equals(StructuredGoalImpl)»
«val structuredGoal=(goal as StructuredGoalImpl)»
\item{«structuredGoal.name»:
«IF !structuredGoal.object.isNullOrEmpty»
    \textbf{Object:} «structuredGoal.object».
«ENDIF»
«IF !structuredGoal.technique.isNullOrEmpty»
    \textbf{Technique:} «structuredGoal.technique».
«ENDIF»
«IF !structuredGoal.quality.isNullOrEmpty»
    \textbf{Quality:} «structuredGoal.quality».
«ENDIF»
«IF !structuredGoal.ptView.isNullOrEmpty»
    \textbf{Point of View:} «structuredGoal.ptView».
«ENDIF»
```

```
	«IF !structuredGoal.contextOf.isNullOrEmpty»
	    \textbf{Context Of:} «structuredGoal.contextOf».
	«ENDIF»
	}

	  «ENDIF»
	    «ENDFOR»
	    \end{itemize}
	    «ENDIF»
	    '''
	def String generateQuestions(Experiment experiment)
	    '''
	    «IF !experiment.researchQuestions.isNullOrEmpty»
	    \section{Research Questions}
	    \begin{itemize}

	    «FOR question:experiment.researchQuestions»
	    \item{«question.description» «IF question.goal!==null». Related
	       to «question.goal.name»«ENDIF»}
	    «ENDFOR»

	    \end{itemize}
	    «ENDIF»
	    '''

	def String generateOverview(Experiment experiment)
	'''

	    «FOR variable : (experiment.researchHypotheses as
	       List<ResearchHypothesis>).map[
	       formula.depVariable].removeDuplicates»
	  <<overview_«variable.name.convert», include=TRUE, echo=FALSE,
	     warning=FALSE , message=FALSE >≥
	  DF <- data_summary( subset( json_data, «FOR object :
	     experiment.objectsInUse BEFORE "(" SEPARATOR "|"»object ==
	     '«object.name»'«ENDFOR») & !is.na(«variable.name.convert» )),
	     varname = "«variable.name.convert»", groupnames =
	     c("treatmentDescription", "objectLabel", "objectOrder"))
	  «generatePlotOverview( experiment, variable)»
	  @
	    «ENDFOR»
	'''

	protected def CharSequence generatePlotOverview(Experiment
	   experiment, DependentVariable variable)
```

```
        '''«IF experiment.objectsScaleType.equals(ScaleType.NOMINAL)»
      DF$objectLabel <- factor(DF$objectLabel, levels = c( «FOR object
          : experiment.experimentalObjects SEPARATOR
          ","»"«object.description»"«ENDFOR»))
        «ENDIF»

        ggplot(DF, aes(x = objectLabel, y = «variable.name.convert»,
            group = treatmentDescription, color = treatmentDescription))
            +
      geom_errorbar( aes( ymin = «variable.name.convert» - sd, ymax = «
          variable.name.convert»+sd), width = .1, linetype = 3) +
      geom_line() + geom_point()+
     scale_color_brewer(palette="Paired") +
     theme_bw() +
    «IF experiment.objectsScaleType.equals( ScaleType.NOMINAL)»
        scale_x_discrete(name = "«experiment.objectsDescription»")+
    «ENDIF»
    «IF experiment.objectsScaleType.equals( ScaleType.ABSOLUTE)»
        scale_x_continuous(name = "«experiment.objectsDescription»",
            breaks_continuous( data = DF$objectLabel, steps = 10))+
    «ENDIF»
    «IF experiment.objectsScaleType.equals( ScaleType.LOGARITHMIC)»
        scale_x_log10(name = "«experiment.objectsDescription»( log
            scale)", breaks_log( data = DF$objectLabel, steps=10))+
    «ENDIF»

    «IF variable.scaleType.equals( ScaleType.NOMINAL)»
        scale_y_discrete(name = "«variable.description» «IF
            variable.unit !== null»(«variable.unit»)«ENDIF»")+
    «ENDIF»
    «IF variable.scaleType.equals( ScaleType.ABSOLUTE)»
        scale_y_continuous(name = "«variable.description» «IF
            variable.unit !== null»(«variable.unit»)«ENDIF»")+
    «ENDIF»
    «IF variable.scaleType.equals( ScaleType.LOGARITHMIC)»
        scale_y_log10(name = "«variable.description» «IF variable.unit
            !== null»( «variable.unit»)«ENDIF»( log scale)")+
    «ENDIF»
    ggtitle("«variable.description» Overview") +
    theme(legend.title = element_blank())
        '''

    def String generateOverview(Experiment experiment, Treatment
       treatment)
    '''
```

```
        <<«treatment.name», include = TRUE, echo = FALSE, warning = FALSE , message = FALSE >≥
        «FOR variable : (experiment.researchHypotheses as List<ResearchHypothesis>).map[ formula.depVariable].removeDuplicates»
      DF = subset( json_data, «FOR object : treatment.experimentalObjects BEFORE "(" SEPARATOR "|" AFTER ")"»object == '«object.name»'«ENDFOR» & treatment == '«treatment.name»')
      DF$objectLabel = ordered(DF$objectLabel, levels = levels(DF$objectLabel)[order( as.numeric( by( DF$«variable.name.convert», DF$objectLabel, mean)))])
      boxplot_«treatment.name»_«variable.name.convert» = ggplot(DF, aes(x = objectLabel , y = «variable.name.convert»)) +
    geom_boxplot(fill = "#4271AE", colour = "#1F3552",alpha = 0.7,outlier.colour = "#1F3552", outlier.shape = 20)+
    theme_bw() +
    scale_x_discrete(name = "Experimental Object")+
    ggtitle("«variable.description» by «treatment.factor.description» for «treatment.description»") +
    ylab("«variable.description» «IF variable.unit !== null»(«variable.unit»)«ENDIF»")
    boxplot_«treatment.name»_«variable.name.convert»
        «ENDFOR»
        @
    '''

    def String generateObjectOverview(Experiment experiment,ExperimentalObject object)
    '''
        <<«object.name», include = TRUE, echo = FALSE, warning = FALSE , message = FALSE >≥
        «FOR variable : (experiment.researchHypotheses as List<ResearchHypothesis>).map[ formula.depVariable].removeDuplicates»
      DF = subset(json_data, («FOR treatment : experiment.treatmentsInUse SEPARATOR "|"»treatment == '«treatment.name»'«ENDFOR») & object == '«object.name»')
      DF$treatmentDescription = ordered(DF$treatmentDescription, levels = levels( DF$treatmentDescription)[order( as.numeric( by( DF$«variable.name.convert», DF$treatmentDescription, mean)))])
      boxplot_«object.name»_«variable.name.convert» = ggplot(DF, aes(x =treatmentDescription , y = «variable.name.convert»)) +
    geom_boxplot(fill = "#4271AE", colour = "#1F3552",alpha = 0.7,outlier.colour = "#1F3552", outlier.shape = 20)+
    theme_bw() +
    scale_x_discrete(name =
       "«experiment.treatmentsInUse.head.factor.description»")+
    ggtitle("«variable.description» by «
       experiment.treatmentsInUse.head.factor.description» for «
       object.description»") +
    ylab("«variable.description» «IF variable.unit !== null»( «
       variable.unit»)«ENDIF»")
    boxplot_«object.name»_«variable.name.convert»

        «ENDFOR»
        @
    '''

    def convert(String depVariable) {
        switch(depVariable){
      case "cpuConsumption":"cpu"
      case "memoryConsumption":"memory"
      default: depVariable
        }
    }

}
```

Appendix D

Empirical Evaluation

In this appendix, we present the execution scripts (Listings D.1 and D.3) and the experiment specifications (Listings D.2 and D.4) used in the empirical evaluation. The complete set of scripts, specifications, and results data are available in the suplementary material.

Listing D.1: Excerpt an execution script used in Experiment 2

```
for size in big small
do
    redis-cli flushall
    for conf in \
        kaluza.nocache.conf \
        kaluza.cashew.conf \
        kaluza.cashew-except-order.conf \
        kaluza.cashew-except-reduce.conf \
        kaluza.cashew-except-remove.conf \
        kaluza.cashew-except-renameAlph.conf \
        kaluza.cashew-except-renameVar.conf
    do
        expsdir=exps.${conf}_$i.output
        mkdir -p ${expsdir}
        for constraint in $(ls ${INPUTDIR} | grep $size)
        do
            runkal ${INPUTDIR}/${constraint} ${conf} >
               ${expsdir}/${constraint}
        done
        echo ''
    done
    for conf in \
        kaluza.nocache.conf \
        kaluza.cashew.conf
    do
```

```
        mkdir -p results
        expsdir=exps.${conf}_$i.output
        grep totalSolvingTime ${expsdir}/big*.smt2 | awk '{print$3}' | st > results/${expsdir}.big.time
        cat results/${expsdir}.big.time
        grep totalSolvingTime ${expsdir}/small*.smt2 | awk '{print$3}' | st > results/${expsdir}.small.time
        cat results/${expsdir}.small.time
    done
    for expsdir in $(ls | grep 'exps.*.output$')
    do
        cat ${expsdir}/big*.smt2 | grep Canonicalized | sed 's/Canonicalized: //' | sort | uniq -c | sort -nr -k1 | tee ${expsdir}.big.orbits | awk '{print$1}' | st > results/${expsdir}.big.orbits
        cat ${expsdir}/small*.smt2 | grep Canonicalized | sed 's/Canonicalized: //' | sort | uniq -c | sort -nr -k1 | tee ${expsdir}.small.orbits | awk '{print$1}' | st > results/${expsdir}.small.orbits
        cat ${expsdir}/*.smt2 | grep Canonicalized | sed 's/Canonicalized: //' | sort | uniq -c | sort -nr -k1 | tee ${expsdir}.all.orbits | awk '{print$1}' | st > results/${expsdir}.all.orbits
    done
done
```

Listing D.2: Excerpt of an experiment specification used in 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 equal to  Average time for No Cache"},
    RH2 {maxTime cashew = noCache description "Maximum time for Cashew is equal to  Average time for No Cache"},
    RH3 {sumTime cashew = noCache description "Total time for Cashew is equal to  Average time for No Cache"},
    RH4 {orbits cashew = noCache description "Number of Orbits for Cashew is equal to  the Number of Orbits for No Cache"},
    RH5 {orbits cashew = cashewExceptOrder description "Number of Orbits for Cashew is equal to  the Number of Orbits for Cashew Except Order"},
    RH6 {orbits cashew = cashewExceptReduce description "Number of Orbits for Cashew is equal to  the Number of Orbits for Cashew Except Reduce"},
    RH7 {orbits cashew = cashewExceptRemove description "Number of Orbits for Cashew is equal to  the Number of Orbits for Cashew Except Remove"},
    RH8 {orbits cashew = cashewExceptRenameAlph description "Number of Orbits for Cashew is equal to  the Number of Orbits for Cashew Except Rename Alph"},
    RH9 {orbits cashew = cashewExceptRenameVar description "Number of Orbits for Cashew is equal to  the Number of Orbits for Cashew Except Rename Var"}
  }
  Experimental Design {
    runs 1
  }
  Dependent Variables {
    averageTime { description "Average time" scaleType Absolute unit "s" instrument averageTimeCommand },
    maxTime { description "Maximum time" scaleType Absolute unit "s" instrument maxTimeCommand },
    sumTime { description "Total time" scaleType Absolute unit "s" instrument sumTimeCommand },
    orbits { description "Number of Orbits" scaleType Absolute instrument orbitsCommand }
  }
  Instruments {
    averageTimeCommand {command  ""  valueExpression "mean:" },
    maxTimeCommand {command  ""  valueExpression "max:" },
    sumTimeCommand {command  ""  valueExpression "sum:" },
    orbitsCommand {command  ""  valueExpression "N-orbits:" }
  }
  Factors {
    transformations { description "Transformations enabled" scaleType
       Nominal}
  }
  Treatments {
    cashew description "All transformations"  factor transformations
       parameters{conf "kaluza.cashew.conf"} execution cashewExecutor,
    noCache description "No cache"  factor transformations
       parameters{conf "kaluza.nocache.conf"} execution cashewExecutor,
    cashewExceptOrder description "Except order"  factor
       transformations parameters{conf
       "kaluza.cashew-except-order.conf"} execution cashewExecutor,
    cashewExceptReduce description "Except removeVar"  factor
       transformations parameters{conf
       "kaluza.cashew-except-reduce.conf"} execution cashewExecutor,
    cashewExceptRemove description "Except removeConj"  factor
       transformations parameters{conf
       "kaluza.cashew-except-remove.conf"} execution cashewExecutor,
    cashewExceptRenameAlph description "Except rename alph"  factor
       transformations parameters{conf
       "kaluza.cashew-except-renameAlph.conf"} execution cashewExecutor,
    cashewExceptRenameVar description "Except rename var"  factor
       transformations parameters{conf
       "kaluza.cashew-except-renameVar.conf"} execution cashewExecutor
  }
  Objects { description "Constraints" scaleType Nominal {
    small {
      description "SMC-Small"
      parameters {
          preffix "small"
      }
    },
    big {
      description "SMC-Big"
        parameters {
          preffix "big"
        }
      }
    }
  }
  Executions {
     cashewExecutor {
      command "/root/phab/green/run-orbits.sh
         ${treatment.parameter.conf} ${object.parameter.preffix}"
      timeout 100000
      preprocessing {
          redisFlush { command "redis-cli flushall" },
      }
    }
}
```

Listing D.3: Excerpt of an execution script used in Experiment 3

```
function jpf() {
    java -Xmx2g -jar ${HOME}/phab/jpf-core/build/RunJPF.jar $@
}
expdir=exps.${expname}
mkdir -p ${expdir}
for flavor in nocache trivialcaching cashew
do
    redis-cli flushall
    echo ''

    for pw in $(cat ${passwordsfile})
    do
        echo "Running ${flavor}.${pw}"
        jpf ${seriesname}.${flavor}.jpf +target.args=${pw} >
           ${expdir}/${flavor}.${pw}.log
    done

    redis-cli save
    cp /var/lib/redis/dump.rdb redis_after.${expname}.${flavor}.rdb

done

for flavor in nocache
do
    sat_time=$(grep ABCService::timeConsumption
       ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor sat_time $sat_time
    count_time=$(grep ABCCountService::timeConsumption
       ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor count_time $count_time
    satpluscount_time=$(grep
       "\(ABCService\|ABCCountService\)::timeConsumption"
       ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor satpluscount_time $satpluscount_time
    satpluscountplusnorm_time=$(grep "::timeConsumption"
       ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor satpluscountplusnorm_time $satpluscountplusnorm_time
    echo
done

for flavor in trivialcaching cashew
do
    sat_time=$(grep ABCService::timeConsumption
       ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor sat_time $sat_time
    count_time=$(grep ABCCountService::timeConsumption ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor count_time $count_time
    satpluscount_time=$(grep "\(ABCService\|ABCCountService\)::timeConsumption" ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor satpluscount_time $satpluscount_time
    satpluscountplusnorm_time=$(grep "::timeConsumption" ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t/1000.0}')
    echo $flavor satpluscountplusnorm_time $satpluscountplusnorm_time
    echo
    sat_hits=$(grep ABCService::cacheHits ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t}')
    echo $flavor sat_hits $sat_hits
    sat_misses=$(grep ABCService::cacheMisses ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t}')
    echo $flavor sat_misses $sat_misses
    sat_hitmissratio=$(python -c "print(float($sat_hits)/$sat_misses)")
    echo $flavor sat_hitmissratio $sat_hitmissratio
    sat_hitpercentage=$(python -c "print(float($sat_hits)/($sat_hits + $sat_misses))")
    echo $flavor sat_hitpercentage $sat_hitpercentage
    echo
    count_hits=$(grep ABCCountService::cacheBoundedHits ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t}')
    echo $flavor count_hits $count_hits
    count_misses=$(grep ABCCountService::cacheMisses ${expdir}/${flavor}.*.log | awk '{t=t+$3}END{print t}')
    echo $flavor count_misses $count_misses
    count_hitmissratio=$(python -c "print(float($count_hits)/$count_misses)")
    echo $flavor count_hitmissratio $count_hitmissratio
    count_hitpercentage=$(python -c "print(float($count_hits)/($count_hits + $count_misses))")
    echo $flavor count_hitpercentage $count_hitpercentage
    echo
done
```

Listing D.4: Excerpt of an experiment specification used in Experiment 3

```
Experiment cashew {
  description "Constraint Normalization and Parameterized Caching for Quantitative Program Analysis"
  Research Hypotheses {
    RH1 {sumTime cashew = nocache description "Total time for Cashew is equal to  Total time for No Cache"},
    RH2 {sumTime cashew = trivialcaching description "Total time for Cashew is equal to  Total time for No Normalization"},
    RH3 {hits cashew = trivialcaching description "Number of hits for Cashew is equal to  Number of hits for No Normalization"},
    RH4 {misses cashew = trivialcaching description "Number of misses for Cashew is equal to  Number of misses for No Normalization"},
    RH5 {hitsMissesRatio cashew = trivialcaching description "Hits/Misses ratio for Cashew is equal to  Hits/Misses ratio for No Normalization"}
  }
  Experimental Design {
    runs 1
  }
  Dependent Variables {
    sumTime { description "Total time" scaleType Absolute unit "s" instrument sumTimeCommand },
    hits { description "Hits" scaleType Absolute  instrument hitsCommand },
    misses { description "Misses" scaleType Absolute  instrument missesCommand },
    hitsMissesRatio { description "Hits/Misses ratio" scaleType Absolute  instrument hitsMissesRatioCommand }
  }
  Instruments {
    sumTimeCommand {command  ""  valueExpression "time:" },
    hitsCommand {command  ""  valueExpression "hits:" },
    missesCommand {command  ""  valueExpression "misses:" },
    hitsMissesRatioCommand {command  ""  valueExpression "hitsmissesratio:" }
  }
  Factors {
    transformations { description "Transformations enabled" scaleType Nominal}
  }
  Treatments {
    nocache description "No cache"  factor transformations parameters{conf "kaluza.nocache.conf"} execution cashewExecutor,
    trivialcaching description "No Normalization"  factor transformations parameters{conf "kaluza.cashew-except-order.conf"} execution cashewExecutor,
    cashew description "Cashew"  factor transformations parameters{conf "kaluza.cashew.conf"} execution cashewExecutor
  }
  Objects { description "Constraints" scaleType Nominal {
      password {description "Password1"},
      password2 {description "Password2"},
      obscure {description "Obscure"},
      crime {description "CRIME"}
    }
  }
  Executions {
     cashewExecutor {
      command "jpf-security/src/examples/cashew/run-security.sh ${treatment.name} ${object.name}"
      timeout 100000
      preprocessing {
        redisFlush { command "redis-cli flushall" },
      }
      postprocessing {
        redisSave { command "redis-cli save" },
     }
    }
  }
}
```