Towards an Infrastructure for Empirical Software Measurement Studies

Eric de Backer
Towards an Infrastructure for Empirical Software Measurement Studies

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Eric de Backer
born in Heesch, the Netherlands

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
Towards an Infrastructure for Empirical Software Measurement Studies

Author: Eric de Backer
Student id: 1158988
Email: E.A.M.deBacker@student.tudelft.nl

Abstract

This thesis introduces Erix, Empirical Research Infrastructure for Clustered Software. Erix supports the setup and execution of software measurement research, a special type of empirical research that conducts and analyzes software measurements. The goal of Erix is to reduce the costs of software measurement experiments by the use of automation. Besides, several other goals have been accomplished: support for replication and aggregation of experiments, reuse of research definitions, use of standard measurement and analysis tools and automated storage of experiment results. Furthermore, Erix has been tested with a case study and it does what it is intended to.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. Ing. L. Moonen, Faculty EEMCS, TU Delft
Co-supervisor: Drs. C. Boogerd, Faculty EEMCS, TU Delft
Committee Member: Dr. M. de Weerdt, Faculty EEMCS, TU Delft
A little more than five years ago, I decided to come to Delft to prolong my study for 3 years with the master Computer Science. Now, 5 years later, I will finally reach this goal. In the past 5 years, I have learned a lot about many things. This thesis is probably the most significant result of my transformation to a true Delft engineer.

Many thanks go to Leon and Cathal for supervising the project, the sensible advice and the time and freedom they gave me to learn. Furthermore, I would like to thank my parents, for supporting me unconditionally. And of course I want to thank Hanneke, for way too many things to mention.

Eric de Backer
Delft, the Netherlands
September 4, 2007
Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
  1.1 Problem Definition ............................................. 1
  1.2 Project Goal .................................................. 2
  1.3 Background .................................................... 3
  1.4 Thesis Outline ................................................ 5

2 Empirical Research in Software Engineering 7
  2.1 An Introduction to Empirical Research ......................... 7
  2.2 Empirical Methods in Software Engineering .................... 10
  2.3 Software Measurement .......................................... 12
  2.4 Problems in Empirical Research ................................ 16
  2.5 The Project Goal Revisited .................................... 18

3 Automation in Empirical Research 21
  3.1 Automated Experiments ........................................ 21
  3.2 Software Measurement Research ................................ 24
  3.3 The Experiment Definition .................................... 27
  3.4 The Use of Automation in Empirical Software Engineering .... 27

4 The Empirical Research Infrastructure for Clustered Software 29
  4.1 Requirements .................................................. 29
  4.2 Technical Design ............................................... 31
  4.3 Implementation ................................................ 39
  4.4 How to Use Erix ............................................... 44
## CONTENTS

5 Erix Experience in a Case Study 51  
5.1 The Inspection Results Prioritizing Experiment 51  
5.2 Lessons Learned 58  

6 Related Work 61  
6.1 Similar Infrastructures 61  
6.2 Additional Information 64  

7 Conclusions and Future Work 67  
7.1 Contributions 67  
7.2 Conclusions 67  
7.3 Future work 69  

Bibliography 71  
A Glossary 75  
B Empirical Software Engineering Summary 77  
C The Nix Expression Language 83  
C.1 A Simple Example 83  
C.2 Values 84  
C.3 Language Constructs 84  
C.4 Operators 85  
D The Prioritizing Case Study ERIX Definition 87  
Appendices Bibliography 97
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A causal model of the operation defects (taken from [23])</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Model of a software measurement research</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Model of a clustered software measurement research</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>The components of the imperative solution</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>The preliminary object model</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Model of a software measurement research including the Nix packages</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>The squatter toolset architecture (from [39], page 8)</td>
<td>62</td>
</tr>
<tr>
<td>6.2</td>
<td>The toolset and its connections (from [47], page 176)</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>Two components, C and D, with required and provided interfaces, respectively</td>
<td>65</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter introduces the problem to solve and goal to reach. This project has been fulfilled as a graduation project at the software research Engineering group, at the department of Software Technology within the Faculty of Electrical Engineering, Mathematics and Computer Science of the Delft University of Technology.

1.1 Problem Definition

Software has been around for over 50 years now, it has been used for various tasks and situations, from simple calculations in a calculator to managing a car’s engine. All this software needs to be developed, operated and maintained. Software engineering studies these processes scientifically: systematic, disciplined and quantifiable [1]. Some examples of experiments in software engineering are experiments to improve software quality and experiments to speed up software development projects.

We focus on empirical research in software engineering. In empirical research, a certain situation or product is observed in order to verify some beliefs of the subject. Empirical research is used in software engineering to validate theories and hypotheses. When we conduct an empirical experiment, we first need a theory to test, then we can observe an appropriate situation, that probably needs to be analyzed subsequently. The result can be used to draw a conclusion for the theory or hypotheses. Empirical experiments can be done for instance with a case study or a survey.

The SWERL department (Software Evolution Research Lab) of the Delft University of Technology studies various aspects of software engineering. Empirical research is used in several of these studies. We focus on the experiments wherein certain properties of software modules are measured by existing software tools, software with the goal to help software engineers. To do this in a systematic manner, dedicated software tools have been developed that coordinate the measurement tools for a specific experiment. The development of these coordination tools costs a lot of effort and the solutions tend to be specific for a particular experiment.

The SWERL researchers believe that the coordination (directing the input and output of the
measurement tools) of the existing measurement software can be generalized into a software measurement infrastructure.

1.2 Project Goal

In the previous section we have identified a problem with empirical experiments. We intend to solve this effort problem by using automation. Unfortunately, not all kinds of empirical research can be (fully) automated. For instance, an interview cannot be taken without human interference. Therefore we need to define a special type of empirical experiments which can be fully automated, we will call this software measurement experiments.

Software measurement experiments can be conducted by using the right tools on the correct input. The input for a certain tool must be delivered by the researcher or generated by another tool. This causes dependencies between the execution of the tools in a certain experiment. These properties (input, tools and dependencies) must be known to define an experiment.

In order to save effort, we want to automate the execution of the experiment by the use of a generic (parameterized) infrastructure. This infrastructure will take an experiment definition and perform the experiment according to the definition. This means that the goal of the infrastructure is to convert the experiment definition to a sequence of actions to run the measurement and analysis tools.

The research goal: Find a method to effectively setup a software measurement experiment and conduct this experiment automatically and properly.

Several initial ideas on the infrastructure have been proposed. They represent various aspects of the infrastructure. These ideas are listed in the upcoming paragraphs.

Reuse of experiment definitions The infrastructure requires an experiment definition to conduct the experiment along with appropriate input. The definition needs to be created by the researcher. As a result, it can save a considerable amount of time and effort if an experiment definition can be reused in other experiments.

Storage of all (interim) results The results of an experiment may be used in other studies than it was originally intended. Furthermore, the results can be used to check the experiment. Therefore, it may be useful to store all the experiment results. Of course, the results are useless if they cannot be found when needed. Consequently, the storage needs to be done consistently and transparent for all experiments.

Built-in analysis procedures The analysis of the experiments that are conducted with the infrastructure may be very similar for many experiments. Therefore, it can be useful to equip the infrastructure with the most used analysis procedures.


**Online and offline experimenting**  The experiment cannot always be conducted in the same environment as the infrastructure. In such cases, the infrastructure cannot control the experiment itself, we call this offline experimenting. This is useful for, for instance, embedded systems.

**OS-independence**  To make the infrastructure available for as many people as possible, it may be useful to make it possible to run the infrastructure on different operating systems.

**The use of the infrastructure**  The use of the infrastructure may be made easier by supplying the user with certain tools. For instance, a tool to create an experiment definition step by step and an easy installation process.

**Proper empirical research**  Empirical research is an upcoming field in software engineering. Therefore, several problems and challenges exist in the execution of empirical experiments (Section 2.4). To conduct proper empirical research, these problems and challenges must be known and solved.

During the development, these ideas will grow to be the infrastructure’s requirements and boundaries. These are used in the development of the infrastructure.

## 1.3 Background

The project goal is to find a method to setup and conduct empirical research for software engineering. This section introduces several fields where the solution for the project goal may be used.

### 1.3.1 Software Tools

The infrastructure that will be described in the upcoming chapters is an example of a software tool. Therefore, a short overview is given of software tools in general.

The definition of software tools states that software tools make the life of an software engineer easier (and probably more enjoyable). A lot of tools have been developed, from a way to coordinate the software build process (e.g., make [19]) to complete software design environments (e.g., Rational Rose\(^1\)).

Sommerville discusses a functional classification of software tools [46], the classes are: planning, editing, change management, configuration management, prototyping, method support, language processing, program analysis, testing, debugging, documentation and re-engineering. This classification is arbitrary but it emphasizes the number of tools that are available.

Harrison, Ossher and Tarr [29] state that the greatest challenge in the field of software tools is the integration of tools, processes, artifacts and views. One of the major reasons of

\(^1\)http://www.rational.com
this is the dependence on the context of the tools, for instance the operating system it runs on.

Harrison, Ossher and Tarr advocate that software should be malleable through its entire life-cycle [29]. Now it is only malleable during the development. This means that the software should be prepared for conditions which cannot be foreseen. Software tools can help to adapt software to the new context.

In this thesis, we describe a tool, the infrastructure. In the context of software tools, we can define the goal of the infrastructure to make the software measurement experiments more malleable.

1.3.2 Software Measurement Programs

It is frequently argued that organizations want more control on the software development process. One of the ways to increase control, is by initiating a software measurement program [41]. The goal of such a program is to get insight in the software development process by gathering and analyzing of data from current projects. The results of the measurements are stored in a database and can be obtained when needed in another project, for instance to make predictions about the implementation process. When more projects have been executed within the measurement program, the database (also called experience base) grows. The management can use the knowledge of the recorded projects in combination with the measurements of the current project to control and steer the current project towards the project goals.

It turns out that software measurement programs are an effective way to gain control of a software engineering project if properly implemented [39, 41, 28].

The infrastructure that we are to develop can be used within a software measurement program. The data collection and storage might be done automatically. This will make it easier to setup and use a software measurement program.

1.3.3 Laboratory Packages

Shull et al. [45] discusses laboratory packages, “a laboratory package describes the experiment in specific terms and provides materials for replication, highlights opportunities for variation, and builds a context for combining results of different types of experimental treatments” [45].

A laboratory package supplies the details needed for replication of an experiment. To do this, several package quality goals and guidelines are defined. These guidelines instruct to include the used artifacts, training materials, time estimates and descriptions of the goal, design and context.

Although the term package might indicate that a laboratory package is a software product, a lab package may contain all kinds of data such as paper documentation and software input.

---

2 Also called a software metrics program
The proposed infrastructure needs a definition of each experiment that is conducted. This definition contains all details that are needed to conduct the experiment. Therefore, the use of the infrastructure makes it easier to create a lab package of an experiment.

1.4 Thesis Outline

This thesis consists of seven chapters, besides this introduction. The results of the literature survey of empirical research are discussed in Chapter 2. Chapter 3 introduces the definitions of the type of experiments we focus on in this thesis. The development process of the infrastructure and the product will be described in Chapter 4. The case study that was conducted to test the infrastructure is discussed in Chapter 5. An overview of several software solutions for (more or less) the same problem will be given in Chapter 6. The thesis is ended with the conclusion, some directions for future work and the project’s contributions to the software engineering research field.

This thesis includes four appendices. Appendix A defines the terms that are used in this thesis. An overview of the experiments of the latest five volumes of the journal *Empirical Software Engineering* is given in Appendix B. Appendix C gives an description of the Nix expression language [14]. The language used to define Nix packages. The second appendix shows the definition of the research of the case study research in Chapter 6.
Chapter 2

Empirical Research in Software Engineering

Recall from the problem definition and the project goals that the goal of the research is to support researchers in conducting empirical research in software engineering. Therefore, the current state of empirical research in software engineering will be described in this chapter.

2.1 An Introduction to Empirical Research

Empirical research tries to answer research questions by collecting empirical observations or data. One of the most famous examples of empirical research is said to be conducted by Galileo Galilei on the tower of Pisa [38] in 1589 where he counter proves a theorem of Aristotle that lighter objects fall less fast than heavier objects. This experiment illustrates the start of modern empirical research. Nowadays, empirical research is a celebrated technique in a broad range of sciences [3].

In software engineering, empirical research is an upcoming field. Many researchers recognize the need for an empirical basis in software engineering [10, 34, 63]. B. Curtis [10] mentioned in 1980 that a lot of important software characteristics are not modeled sufficiently to do correct research. Surprisingly, 20 years later, in [63] it is noted that “software engineering research has failed to produce the deep models and analytical tools that are common in other sciences” because “We do not know the fundamental mechanisms that drive the costs and benefits of software tools and methods”. More empirical research is needed to find these fundamental mechanisms [63].

The use of empirical research is diverse in software engineering, for instance, validation [53] and prediction [22].

1Although this “leaning tower experiment” is one of the most famous experiments ever, it has never taken place [38].
2.1 An Introduction to Empirical Research

2.1.1 Qualitative or Quantitative

Empirical research can be partitioned in two groups: quantitative research and qualitative research [44, 52].

Quantitative Research

Quantitative research is based on the quantification of the properties of the subject under investigation. The quantification can for instance be done by measuring the properties. Quantitative research can be used to fulfill the following goals [15, 48]:

- Describing the current state of the world. Subsequent measurements can be used to discover trends and patterns. For instance, the amount of manpower can be measured during the development of certain functions. This information can be used to identify functions that are difficult to develop.

- Predicting project properties. The data described in the previous point can be used to predict project properties such as costs. Section 2.3.2 gives a short introduction to the process of prediction.

- Checking requirements and goals. Every project has its requirements and goals. To verify if these requirements are met, they must be expressed quantitatively. Quantitative research can be used to identify appropriate properties and propose required values.

- Monitoring project progress. The progress of the project can be indicated by measurements.

- Analyzing costs and benefits. From the managers point of view, the cost-benefit relation is important to make future decisions. Quantitative research can supply the needed numbers.

Qualitative Research

Qualitative research has been introduced to deal with the “people-factor” in software engineering, the non-technical issues. Qualitative research is used to take complex factors into account, e.g., organizational issues, management issues. There are several differences between qualitative experiments and quantitative experiments [44]: (1) the representation of the results is in numbers for a quantitative experiment and in pictures or words for a qualitative experiment. The results of qualitative experiments are more informative but they are considered “softer” and “fuzzier”; (2) the researcher is forced to delve into the complexity of the problem when conducting a qualitative experiment in contrary to a quantitative experiment where the complexity is abstracted; (3) a qualitative experiment is more labor intensive than a quantitative experiment; (4) the results of a qualitative experiment are more difficult to summarize or simplify.
A method of data collection in qualitative research is the interview. Several types of interviews can be distinguished, such as structured and unstructured interviews. In a structured interview, the interviewer asks the interviewee predefined questions. Quantitative interviews (such as the interview mentioned in section Section 2.2) are structured interviews with quantifiable answers (e.g., yes/no, high/medium/low). In an unstructured interview, not only the answers but also the questions may not be known beforehand, the interviewee can give both the (open-ended) questions and the answers. The task of the interviewer in an unstructured interview is to keep the interviewee close to the subject. A property of an unstructured interview is that it can deliver unforeseen results and directions. Most qualitative interviews are semistructured to combine the advantages of both interview types.

The goal of the project is to automate empirical experiments. From the definition, we see that qualitative research is hard to automate. Therefore, in the rest of the thesis, we focus on quantitative research unless it is mentioned explicitly.

2.1.2 Examples

Next, two examples will be introduced which will be used as running examples throughout this thesis:

**Example 2.1** Prioritizing code inspection results

Boogerd and Moonen [6] compare static profiling to dynamic profiling. The goal of this experiment is to investigate the use of static profiles to prioritize software inspection results [17].

Profiling is a technique to determine the execution path of a software component. Profiling can be done in two distinctive ways: statically and dynamically. Dynamic profiling is done by running the software under investigation and record the execution path. In contrary, in static profiling, the execution path is reconstructed without running the software. This reconstruction process can be done with the use of branch prediction heuristics.

Boogerd and Moonen found that statical profiling might be used to prioritize software inspection results.

**Example 2.2** Checking the heuristics

Example 2.1 uses branch prediction heuristics to calculate the execution likelihood. A branch heuristic represents the probability to take a certain branch, for instance in an if-statement. An if statement has two branches: the target branch, this is taken in case the if-condition evaluates to true and the fall-through branch, this branch is taken if the if-condition evaluates to false. The heuristic in this example represents the probability that the successor branch is taken.

To give an idea what a heuristic looks like, the definition of the return heuristic [2, 11] is given: *Predict the successor that contains a return is not taken* The heuristic contains two parts, a selection property and a predictor. In the return heuristic the selection property is “the successor that contains a return”, and the predictor is “not taken”. So if one of the branches contains a return statement, then that branch is probably not taken.
The heuristics that are used in [6] are taken from arbitrary software, it is imaginable that the software domain or individual factors influence these heuristics. For instance, a programmer can have the habit to write returns in branch which are taken. This could influence the accuracy of the return heuristic.

A suitable empirical study based on heuristics would be to calculate and compare the heuristic probabilities for a certain domain, company, programmer, etc. If the heuristic value vary much, it is questionable if general heuristics can be used in Example 2.1 on Page 9.

### 2.2 Empirical Methods in Software Engineering

There are several methods for conducting empirical research in software engineering. Three are mentioned by [24, 63]: Controlled experiments, case studies and surveys. In [52], besides these three, a fourth method is mentioned: post mortem analysis.

The difference between these methods is mainly in collection of the data and the analysis of this data. Each empirical study should contain the five steps [63], depicted in Table 2.1.

<table>
<thead>
<tr>
<th>Step</th>
<th>Example 2.1 on Page 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Formulate the hypotheses or questions.</td>
<td>The statically determined execution likelihood is close enough to the dynamically determined execution likelihood to use it for fault ranking.</td>
</tr>
<tr>
<td>2. Observing a situation.</td>
<td>The SUAs (Software Units under Analysis) are examined to extract its static profile and the SUAs are dynamically profiled.</td>
</tr>
<tr>
<td>3. Abstracting the observations into data.</td>
<td>The static and dynamic execution likelihoods are calculated from the profiles for every SUA.</td>
</tr>
<tr>
<td>4. Analysis the data.</td>
<td>Compare the statically extracted execution likelihood to the dynamically extracted execution likelihood for the same SUA.</td>
</tr>
<tr>
<td>5. Drawing conclusions.</td>
<td>If the different likelihoods are close enough then accept, otherwise reject.</td>
</tr>
</tbody>
</table>

Table 2.1: The steps of an empirical study

The empirical methods: controlled experiment, case study, survey and post mortem analysis are described in the next sections.

#### 2.2.1 Controlled experiment

In a controlled experiment the researcher can control one or more variables, the independent variables. These variables, together with variables which cannot be influenced but affect the result (confounding variables), determine the outcome (dependent variables). By fluctuating the input and registering the outcome of the experiment, a conclusion can be drawn about the effect of the input on the outcome of the experiment.
The research in Table 2.1 is an example of a controlled experiment. The methods used to determine the execution likelihood are the independent variables and the execution likelihoods for the same SUA are the dependent variables. The compiler that is used to compile the SUAs for the dynamically determined execution likelihood is a confounding factor because, for example, some compilers precompile some of the if statements, which are then not counted during profiling. In this experiment, the independent variables, the calculation methods, are mapped onto the execution likelihood.

An important property of a controlled experiment is the validity. The validity indicates how valid the results are, this should be assessed before the presentation of the results. There are 4 types of validity:

- Internal validity is concerned with factors which may influence the dependent variables without the researcher’s knowledge (confounding factors).
- External validity indicates the ability to generalize the results of the experiment.
- Conclusion validity indicates the probability that the conclusion is correct regarding the relation between the independent variables and the dependent variables.
- Construct validity is concerned with the relationship between the measurement and the theories and concepts behind the experiment.

2.2.2 Case study

In a case study, a single entity (case) is observed during a specific time frame. The independent variables represent a typical situation. Therefore, the results of a case study do not have to be scaled up in size and complexity. In software engineering, case studies are frequently used for evaluating widespread process changes, such as a new project managing method. The result of case studies can be determined by comparing them to the baseline, this is a sister project with another set of variables. In the case of an project managing method, the former project managing method can be used in the sister project.

Because a case study is an observation of only one situation, it is important to minimize the effects of confounding factors. Otherwise, the confounding factors might change the case study results.

2.2.3 Survey

A survey is an investigation to the opinion or experience of people. This can be done by sending questionnaires or interviewing people. The purpose is to learn to know and understand the population’s opinion (and view) on the subject.

2.2.4 Post mortem analysis

Post-mortem analysis is a combination of a case study and a survey. The analysis is about the past and concentrated on one case. Post mortem analysis can be conducted in several ways, such as interviews or analysis of the project documentation.
2.3 Software Measurement

Conducting empirical research often involves taking measurements. In software engineering, we can take measurements from processes (project length for instance) or products (the size of the software).

2.3.1 Measurement Fundamentals

This section is based on [20, 24, 37], however, the information in this section can be found in almost every book about software measurement.

Measurement is the process of quantification of properties (attributes) of real world objects (entities).

For instance, if we want to quantify the height of a person then the person is the entity and the height is the attribute. The definition of a measurement model is needed to reach consensus about the measurement. A measurement model reflects a particular viewpoint of the measurement. In our example, the model can specify if the person is allowed to wear shoes during the measurement.

It is very important that measurement activities have clear objectives [4, 20]. Because, if the measurements are taken without understanding the goals it can never be claimed that the goals are reached.

The representation of measurement is an important topic in the theory of measurement. This representational theory of measurement will be explained with three key words: empirical relations, representation conditions and scales.

**Empirical relations** Empirical relations are relations between entities based on the empirical properties (measures) of an attribute, an example is the relation “taller than” which is based on the attribute height.

**Representation conditions** The mapping of an attribute onto a numerical relation system (the measure) has to preserve the empirical relations. So if person A is taller than person B and the taller than relation is mapped on the numerical > relation then must count: M(A) > M(B).

A common failure in software measurement studies is that they fail to ensure this representation condition [20].

**Scales** There exist five different scales or scale types for attributes: nominal, ordinal, interval, ratio and absolute. These scale types reflect the richness of the empirical relation system of an attribute where nominal is the poorest and absolute is the richest. If the scale type of an attribute is known, then it is possible to determine if a statement about the attribute is meaningful. For example, it is meaningful to state that A is twice as tall as B. This is a meaningful statement because the persons height has the scale type ratio. On the contrary, it is not meaningful to say that software failure X is twice as critical as failure Y because the criticality of a fault has the scale type ordinal. Attributes from the ordinal scale cannot be multiplied.

---

2 One can start a discussion if software is a real world object
3 M(A) is the mapping of the height of person A to a numerical relation system (such as N)
2.3.2 Software Metrics

The research in software measurements is mainly aimed at the development and use of software metrics. Software metrics express characteristics of software (process or product) in numbers. These metrics are being used for prediction, for instance, what a project will cost. A second use of software metrics is assessment, for example, software metrics are used to test if the quality of a software product is high enough.

In the previous decades, many software metrics have been developed. Unfortunately, the development of the metrics itself is emphasized instead of the use of the metrics. This has lead to the extraordinary situation that a lot of proposed software metrics have never been used in a real project [43].

Some confusion exists about the terms software metrics and software measures. These are being used mixed up in various papers. [20, 37] state that there is an important difference between the meanings of the two terms: A software metric is either a measure or a prediction system, so a measure is a metric but the opposite is not true.

Measurement for assessment is done directly or indirectly. When conducting direct measurement, only the involved attribute is measured. For instance, the height of a person can be measured directly with tape measure. In indirect measurement the measure originates from one or more other measurements (with calculation). An example of an indirect measure is density, that is the mass per unit volume. To measure the density, both the volume and the mass should be measured. The density can be calculated by mass / volume.

The second type of use of software metrics, prediction, can be done with a prediction system. A prediction system consists of (1) a model and (2) procedures to predict the model parameters and (3) result interpretation [20]. To predict the project costs, we have a certain project cost model, which needs some parameters, these parameters can be, for example, the amount of use cases and predicted lines of code.

Metric Validation

The two types of software metrics, prediction metrics and assessment metrics, each need to be validated in their own way. A prediction system is valid if the predicted attributes are accurate. A measure is valid if it realizes the measurement theory.

A prediction system can be validated using empirical research. Hypotheses state that the prediction system predicts the attribute within a margin of error. With the empirical analysis methods of Section 2.3.3 the hypotheses (thus the validity of the prediction system) can be confirmed or rejected.

Measure validation is checking if the numerical representation of the attribute is correct. This means that the representational condition is satisfied for the attribute.

There is an implicit assumption that each measure should be a component of a valid prediction system to be valid [37, 20]. The Grubstake Group\(^4\) rejects this assumption because a false prediction can have more reasons than only the inaccurate measure(s). The fact that

\(^4\)The Grubstake Group was a group of researchers of various universities who cooperate in software measurement research. The researchers are: Norman Fenton (City University, London), Robin Whitty (CSSE, South Bank Polytechnic, London), Jim Bieman (Colorado State University), Dave Gustafson (Kansas State University), Austin Melton (Kansas State University), and Albert Baker (Iowa State University).
the project costs were higher than predicted does not mean that the amount of use cases was wrong.

**Example 2.3 Software size**

The size of software seems a fairly easy measurable attribute. Though, there are dozens of metrics which measure one or more aspects of software size: length, complexity and functionality [37]. Which metric is best, depends on the intended use. For instance, if the goal is to predict the number of pages that are needed if the code is printed, the lines of code metric (LOC) is sufficient. The number of printed pages can be predicted with \( \text{LOC} / 50 \), if 50 lines can be printed on one page. Nevertheless, if the goal is to predict the amount of effort needed to re-factor the code, a combination of the three aspects will give the best estimation.

Even if it is obvious which metric we should use (e.g. the number of printed pages prediction), still some question can be raised on how to measure. For instance, to save pages, we decided to not print the blank lines and comments. The measurement model should adapt this change in order to return the right amount of pages.

### 2.3.3 Analysis

The analysis of an empirical research is used to transform the measurement results into useful data. What useful data is depends on the research. In the case of an experiment, data is useful if it specifies to accept or reject the hypothesis. How this can be done, strongly depends on design decisions made earlier in the research. In the two upcoming sections, two of the statistical analysis methods will be described. The analysis methods have been divided in two categories, Bayesian analysis and classic analysis. These two types of analysis approaches originate from the field of statistics.

Note that the type of analysis cannot be chosen without consequences for the rest of the research. The design and the data collection must be done with the goal to feed the chosen analysis method. Therefore, the analysis method must be chosen during the research design.

**Classical Statistical Analysis**

The classical statistical analysis (or frequentist analysis) **only** depends on the available data (from the data collection phase). This data is used to create new knowledge about the subject (inference). An example of classical statistical analysis is the experiment to an uneven coin: we want to know the probability of heads and tails for a certain uneven coin. The most intuitive way to determine this is to throw the coin many times (say 1000 times) and register the outcome of every throw (heads or tails). The probability that the next throw is tails can be calculated using \( \frac{\text{number of tails}}{1000} \).

The classical statistical analysis methods can be divided in two groups: The parametric methods and the non-parametric methods [5]:

- Parametric methods: The goal of the parametric methods is to estimate a parameter for a certain known distribution. If the distribution is determined, the parameter can
be calculated or derived from graphics. An example of the parametric test is the student’s t-test [25].

- Non-parametric methods: The non-parametric methods do not rely on knowledge about the underlying distribution. As a result, these are less accurate than the parametric methods. An example of a non-parametric test is the Kolmogorov-Smirnov test.

**Bayesian Analysis**

Bayesian analysis is also based on statistics. However, instead of using only the collected data, Bayesian analysis depends on a prior distribution for the unknown variable. The prior distribution must be determined without the measured data, for instance it can be specified by an expert. When the prior is specified, the measured data is applied to the prior distribution. This means that the measurement data is combined with the prior data. During this process, the distribution function transforms towards the real distribution. When all data is applied, we have the posterior distribution. Basic information about Bayesian inference can be found in [7] among others.

Chulani, Boehm and Steece [9] give an example of the use of Bayesian analysis in empirical software engineering. The formula $\text{Posterior} \propto \text{Sample} \times \text{Prior}$ is used to calculate the posterior distribution. To determine the prior distribution, eight experts have given an estimate of some cost values.

Another method of Bayesian analysis has been shown in [23], Bayesian Belief Networks (BBN). Bayesian belief networks are graphical networks representing the causal model of a certain situation (a clear explanation of BBNs is given in [8]). An example of an causal model is shown in Figure 2.1. A causal model is the basis for a BBN. To create a BBN from a causal model, probabilities need to be assigned to the leafs of the model (in this figure: problem complexity, design effort, testing effort and operational usage) and conditional probabilities to the relationships. With this additional information, the probabilities of the other events can be calculated.

![Figure 2.1: A causal model of the operation defects (taken from [23])](image)
Causal models have some additional concepts which are missing in classical approaches [23]: (1) BBNs can handle diverse process and product variables; (2) empirical evidence and expert judgement can be used; (3) causal relationships can be modeled; (4) uncertainty about events can be modeled; (5) BBNs can handle incomplete information.

Although Bayesian analysis and Bayesian belief networks are promising techniques in empirical research, they are not used regularly. Therefore a statistician should be consulted to use these approaches [33].

### 2.4 Problems in Empirical Research

It has been mentioned before that empirical research in software engineering still must evolve to become as significant as in other fields of science. This section describes the current problems and challenges. In addition, it introduces several guidelines to prevent the problems and meet the challenges.

#### 2.4.1 Problems

Several problems in empirical research in software engineering have been identified. Below, the most important problems are listed.

- **P1:** The conducted experiments suffer poor statistical design. This is due to poor training of experimental design, statistical analysis and measurement principles in computer science and software engineering [21]. One of the consequences is that the statistical power of most of the experimental software engineering studies is too low [16].

- **P2:** Although indirect measures may be needed, it is important that the relationship between the actual measured property and the indicated property is clear [34]. Otherwise the wrong conclusions can be drawn.

- **P3:** The most important step of conducting an experiment is also the least well done: drawing conclusions [63]. Although there is a batch of data, the study fails to explain why the data is as it is, it does not answer a question. One of the causes of this problem is that the studies lack hypotheses. If we do not clearly define the questions, no answer is good enough.

- **P4:** Empirical research is used mostly for validation of research results. This is a good thing, however, empirical research results can be combined with other knowledge to identify and justify new research. This way, empirical research drives the research [63].

#### 2.4.2 Challenges

The problems of the previous chapters can be abstracted to a few challenges for software measurement studies.
Create better studies The main challenge is to create better empirical studies [63]. To do this, some sub challenges are introduced [63, 21]:

- Striving to causal, actionable and general properties of the research. Causal means that the reasoning follows the if `<condition>` then `<action>` pattern. For example: if the software is large then maintenance will cost much effort. The condition (the software is large) should be actionable, so the researcher can toggle the property, thus conduct the experiment with large software and small software. If the experiment is general then the conclusion can be generalized to a large group of entities.

- Reducing the costs of empirical research. A possible solution is to conduct the experiment (partially) automated. However, automated experimentation is still expensive [21]. This is because: (1) the reuse of artifacts is not easy because artifacts are not always complete, (2) the organization is not standardized (e.g. the directory structure) and (3) artifacts need manual handling.

- Experiments need to be re-done and re-checked by others [63] (or replicated and aggregated [21]). To support replicability, it is needed to establish control on experimental factors and context. This offers some challenges on its own: artifacts are seldom homogeneous (e.g. variation in software versions), the provided detail varies for artifacts (e.g. documentation) and the original experiment is not reported in sufficient detail. Solitary experiments have limited validity. To generalize the findings, families of experiments should be conducted and the findings should be aggregated. The support for aggregation is highly correlated with the support for replication across experiments. To support aggregation, the researcher should be more aware of the experimental context, this should be captured systematically [42].

- Obtaining sample representatives. The sample should represent the whole population under investigation, this is difficult because the sample size is limited and samples may be biased.

Credible interpretations A correctly conducted experiment is only half the process of delivering a correct study in software engineering. The other half is the interpretation of the research and the results. The correct interpretation of the research does not appear to be a challenge for the producer of the research but for the consumer. However, the producer can support a correct interpretation:

- Conduct experiments with high validity. The results are interpreted more safely if the validity is higher. The type of validity that is important for the reader is conclusion validity.

- Support the conclusions. Even if no causal relationship can be proved, discredit other options by other experiments. By the discrediting of some of the other options, the reader can probably understand the conclusion better.
2.4.3 Guidelines

To fulfill the challenges of the previous section, Kitchenham et. al. formulated some preliminary guidelines for conducting empirical research in software engineering [33]. The guidelines are divided in six categories. We list the categories and briefly describe the important guidelines for software measurement studies.

- **The context guidelines** instruct to record as much context as possible. All involved entities, attributes and measures should be defined clearly. It is also important to define the hypotheses or questions and the implications before performing the study. Another context guideline orders to describe similar research and the relationship with the current research.

- **The design guidelines** are mainly concerned with the selection of SUAs out of a population and assigning treatments to them. It is important to define the population and the treatments. Another design guideline dictates to use simple well documented statistical designs, otherwise the analysis can become too complex.

- **The data collection guidelines** prescribe (1) to define the used measures well enough to replicate (repeat) the experiment and (2) to describe the data control methods. The measures have to be defined because software measures are not standardized. The data control methods are used to ensure the accuracy and completeness of data collection. These methods should be described to provide evidence that the data collection is conducted correctly.

- **Analysis guidelines** order to check the data before feeding it to statistical software. First, it should be checked for outliers (sensitivity analyzes). Then, the data should be verified following appropriate quality control procedures.

- **The Presentation guidelines** instruct to describe the used statistical procedures and the statistical package(s) which produced the statistical results. Besides, the results must be presented in an appropriate level of detail and, if possible, the raw data. Furthermore some descriptive statistics must be provided to give the reader of the study a ‘feeling’ of the experiment, for instance the number of SUAs. The final presentation guideline is about graphical representations, these should make the results easier to understand without misleading.

- **Interpretation guidelines** order to draw correct conclusions from the data. It is important to know to which population the conclusions apply and how strong the conclusions are. Finally, it is important to mention the limitations of the study.

2.5 The Project Goal Revisited

The project goal demands that experiments are conducted automatically and properly. The issue on how to conduct an experiment properly is addressed in the previous section: meet
the challenges and follow the guidelines. Therefore, the solutions and rules that are dis-
cussed in the guidelines may be used in the infrastructure. This will be elaborated in Chap-
ter 4. The upcoming chapter will address the type of research that can be automated. We
will see that some experiments cannot be automated at all or just partly. Furthermore, not all
challenges can be fulfilled automatically and not all guidelines can be automatically applied.
Therefore, Chapter 3 will discuss the limits of automation in empirical research.
Chapter 3

Automation in Empirical Research

We want to reduce the costs of empirical research by using automation. This chapter discusses the possibilities and restrictions of automation in this context. Furthermore, the definition of an automated experiment will be discussed briefly and an overview of the use of automation in empirical software engineering will be given.

3.1 Automated Experiments

Recall the project’s main goal: find a method to effectively setup a software measurement experiment and conduct this experiment automatically and properly.

To demonstrate that automation is a decent solution for improving empirical research in software engineering, the upcoming section lists the challenges of Section 2.4. We address the question if the challenges can be fulfilled using automation.

3.1.1 Improve Empirical Research

Causal, Actionable and General Research  The first subgoal is to deliver research that is causal, actionable and general. Recall that causality implies that the experiment is controlled, this means that the input and the outcome must be correlated. If an causal relationship can be controlled, it is called actionable. And an experiment is general if it applies to a wide range of contexts.

These three needed properties for empirical research are difficult to accomplish. The answer for the question if an experiment is causal and actionable can be determined by empirical research. How general the research results are depends on the validity of the experiment. Recall that there exist four kinds of validity: internal validity, external validity, conclusion validity and construct validity. The required validity needs to be determined for every experiment.

This subgoal cannot be accomplished automatically. To test whether an experiment is causal and actionable, separate experiments are needed. The researcher should take this challenge into account when designing the experiment.
Cost Reduction  The costs of empirical experiments are high. It takes much effort to go through the different stages of empirical research. These high costs prevent the use of empirical research in software engineering, disregarding the advantages. Note that this challenge coincides with the project goal.

The choice for an automated solution implies a cost reduction itself, handling something automatic usually costs less effort than handling something manually. However, if the effort needed to define the research is more than the effort needed to setup the research manually, there is no cost reduction (unless the definition may reused). Another feature which could reduce the costs is the reuse of parts of the research definition. The experiment definitions can be used again with other input.

The steps in an empirical research; hypotheses, observe, abstract, analyze and conclude, cannot all be automated. The hypotheses and the conclusion steps cannot be done automatically because these need manual, cognitive, actions of the experimenter. Also, the observation, abstraction and analysis steps cannot be done automatically in every situation, it depends on the experiment that is conducted. For instance, an interview cannot be conducted automatically. Section 3.2 defines two types of research that can be automated.

Support Replication and Aggregation  Replication of an experiment is a repetition of the experiment to verify or broaden the applicability of the results. The replicated experiment can have other input or variables. The replication process can be difficult, the reason for this is that it is difficult to copy the original environment. Shull et al. [45] introduce the tacit knowledge problem in replication, this represents the problem of non-explicit context-details and knowledge of the original experiment. For correct replication, these context-details and knowledge should be captured. Software engineers become aware replication is an important technique in empirical software engineering [45].

Aggregation is a slight variation of replication. Replication is done to verify and broaden the results. Aggregation is used to generalize the research results [21].

The terms aggregation and replication seem to be used interchangeably. In some some literature, the replication is just the repetition of research to verify the original results [21]. Others have broadened the term to include the meaning of generalization the research [45], where [21, 31] think that it is aggregation.

In the description above it seems that replication and aggregation can be supported by registering all kinds of attributes and context of the original research. This means that all these attributes must be recorded and stored in order to use them at the replication or aggregation. Shull et. al. [45] propose the use of laboratory packages for the recording and storing these attributes. The laboratory package’s (or lab package) goal is to supply the replicator with all the information needed to replicate the experiment. The lab package should consist of software, documents, estimates and descriptions [45].

The support of replication and aggregation comprises three tasks: determine the attributes that need to be registered, registering and storing the important attributes. The determining cannot be done automatically because it is a part of the experiment design. The design of an experiment cannot be done automatically. The registering can be done automatically if the attributes expose themselves within a computer environment. The stor-
Obtain Sample representatives  The sample representatives (in our case, guinea pigs) should represent the entire population they are selected from. Do, Elbaum and Rothermel [21] state that two issues are involved in selecting samples: (1) only small samples are possible and (2) the samples are biased.

To select the sample representatives correctly, an overview of the entire population is needed. This is necessary to determine the representativeness and the bias of the sample.

Although it theoretically can be automated, it would cost extra effort to give a selection system the opportunity to select the samples with the right properties. The selection properties should be added manually to the samples. This can be a large amount of effort if one needs to be added for all available samples. Another issue is that the repository of samples should contain a large amount of samples to select from and to get an overview of the entire population.

3.1.2 Improve Interpretations

To improve the interpretations of the research results, two challenges have been proposed: (1) deliver high validity research; (2) support of the conclusions.

High validity research instructs the researcher pay attention to the four types of validity (Section 2.2.1). This is important for the interpretations of the study because the reader is less likely to misinterpret the conclusions.

Support of the conclusions is important for the researcher if no causal relationship can be proved. It is important to describe all tests and experiments to discredit other options. This way, the reader can draw the same conclusions from the data.

Both challenges cannot be supported automatically, the validity of the research is strongly correlated with the research design. The support of conclusions challenge is about the publication of the research results. This is a choice of the researcher, therefore, it cannot be done automatically either.

3.1.3 Simplify Research Setup and Execution

The possibility of meeting each challenge using automation has been tracked in the previous sections. The challenge that are possible to meet using automation are: cost reduction, replication and aggregation support. To reach the other challenges, the experimenter should follow the guidelines of Kitchenham et al. [33] can be used.

We have argued that attributes within an experiment can be registered automatically if they are exposed within a certain environment. The upcoming section defines a specific
3.2 Software Measurement Research

We have determined that not every experiment is suitable for automation. This section defines the type of experiment that can be automated.

**Definition 3.1** A software measurement sequence is a sequence of software measurements and analysis on a single piece of software. The single measurement and analysis task must be able to be fulfilled using one or more shell commands.

A software metric is closely related to the software measurement sequence. The difference is that the goal of a software measurement sequence is not to deliver a single attribute of the software, it can deliver several attributes at once or something else like the length of the output of the software given a certain input. Another difference is that the software measurement sequence demands that the task must be fulfilled using shell commands.

The software measurement sequence is not feasible for all empirical research, but only in the research in which a single piece of software is investigated. Therefore we define a special type of research: software measurement research.

**Definition 3.2** Software measurement research is empirical research to investigate software using a software measurement sequence.

Software measurement research is empirical research with two restrictions: (1) A single piece of software is investigated and (2) The data collection and analysis is done with a software measurement sequence.

The following experiment is an example of software measurement research: The hypothesis is that there are at least 5 bugs per 100 LOC. This can be examined by measuring the total amount of bugs and the lines of code. The outcome can be calculated by \( \frac{number \ of \ bugs}{number \ of \ 100 \ LOC} \). The analysis is the comparison with the hypothesis. So if the outcome > 5 then the hypothesis is accepted, otherwise it is rejected. Note that the validity of this experiment is low, the experiment should be aggregated often to claim that the hypothesis is true.

In comparison to empirical research in general, software measurement research reflects some restrictions about the experiments. This has some consequences for the applicability of the empirical problems, challenges and guidelines of section 2.4. To illustrate this, we give a short example:

The first guideline of Kitchenham et al. [33] instructs to record the experiment context as much as possible. This can be hard for some kinds of empirical research, for instance when people need to be interviewed. In software measurement research, it can be defined more clearly because there is less context which could possibly influence the measurements.
such as emotions in the interview example. Some of the research is even deterministic, this means that if the same research is conducted, the same results will be found.

The definition of software measurement research states that the goal is to obtain some property of the software through software measurement. It might be odd to anchor the method to collect data in the definition. This can be understood by thinking of the goal of the project: find a way to easily define and conduct empirical research. The first remark that we can make is that some empirical research cannot be conducted automatically. For instance, an interview cannot be taken automatically. Therefore we restrict the type of research to the types which can be done automatically: case studies and experiments. However, the data collection process in these types of research may also need manual actions for some experiments. That is why the data collection process is also restricted to software measurement. This property of software measurement research guarantees that the research can be conducted automatically.

Software measurement research consists of several components, these are listed below:

- **Input.** The input of the experiment (for instance a SUA, Software Unit under Analysis). The input is supplied by the experimenter, which input is needed depends on the experiment that is being conducted.

- **Output.** This is the result of the commands of the software measurement sequence.

- **The experiment steps.** The steps which together form the flow of the experiment. These steps each have their own input and output which need to be connected to (1) other steps, (2) experiment input / output and (3) constants. An experiment step can represent a software metric or an analysis.

- **Experiment model.** The experiment model defines the steps and the flow of data between the steps. The research model is also an research step, this makes it possible to use a research model in another research model.

The components can be split up into two layers: the experiment layer and the model layer. The experiment layer contains the input and the output and a “link” to the model layer. To use the model, only the input has to be defined in order to conduct the experiment. The result of the experiment will be returned to the highest layer. The model layer contains the experiment model. The model contains the experiment steps and the connection between them, this is all what is needed to conduct the experiment. This division in two layers makes it possible to reuse the model for other input. In Figure 3.1 on Page 26 the model of an imaginary software measurement experiment is shown with the two steps, two inputs and one output.

**Definition 3.3** Clustered software measurement research is one software measurement sequence applied to more than one piece of software. The goal is to find some property (or properties) of the group of software.

Clustered software measurement research is an iterative execution of the same software measurement sequence on a list of software units under analysis (SUAs). The outcome...
of the individual software measurement research can be analyzed. If the analysis step is omitted, the experiments returns a list of the individual software measurement results. The latter case is a number of software measurement experiments. This is shown in Figure 3.2 on Page 26.

The experiment that is conducted in [6] is an example of a clustered software measurement research. The research described in Example 2.1 on Page 9 is conducted on several SUAs. In this paper, the results of the individual experiments are not analyzed further, the results are presented for each SUA separately.
The results of the separate experiment can be analyzed to “combine” them. Several actions can be applied:

- Summarize, this can be done to give one result of the experiments. Summarizing statistics such as the average and the mean can be used to do this.

- Presentation. The gathered data can be presented in many ways, such as table or a chart.

The three definitions above describe the entire field of empirical research that needs to be covered in the project.

### 3.3 The Experiment Definition

The components of an empirical research have already been identified: input, output, experiment steps and the experiment model. This section specifies the information that is needed to define these components.

**Input:** Two types of input can be distinguished: SUAs and variables. However, SUAs can be specified using a variable as well: the SUA location. Therefore the input is not more than a list of variables. These variables can hold all kinds of data that is needed during the execution of the experiment, such as parameters and paths to SUA input.

**Output:** The user must be able to find the output of the research when the research is finished. To do this an output location can be specified or the infrastructure returns the output location after the research is done.

**Experiment steps:** The experiment steps are responsible for the actual execution of the actions. The experiment step definition should contain all actions. All necessary variables, such as run parameters, need to be defined as well.

**Experiment model:** Recall that the experiment model contains the experiment steps and the dependencies between them. The definition must contain the step definitions or links to the step definitions. Furthermore the dependencies between the steps must be defined.

### 3.4 The Use of Automation in Empirical Software Engineering

We explored the latest five volumes of the journal *Empirical Software Engineering*. This was done for three reasons: (1) to get an overview of current state of the empirical experiments; (2) to identify automation methods and tools; and (3) to explore the need for a general infrastructure to assist to empirical experiments. The journal Empirical Software Engineering was chosen because of the close link to empirical research. Therefore, many empirical experiments have been described in this journal.
The methods used in Empirical Software Engineering  In total 90 papers have been evaluated, these evaluations are listed in Appendix B. It is remarkable that although the papers are in the same journal with the overall subject of empirical research, the subjects of the individual papers are very diverse.

The majority of the papers describe a quantitative experiment, only a few use qualitative research or do not describe an empirical experiment at all.

Automatable Research  The empirical research in the papers has been evaluated on automatability. An experiment has been classified as automatable if the experiment phases, observation, abstraction and analysis can be done without human interference. We have encountered that especially the observation phase often requires human actions, the abstraction and analysis can be done automatically for almost every quantitative experiment. So all experiments with interviews and questionnaires are qualified to be not automatable. Other experiments use data sources that already exist in a appropriate format such as error databases for a product. This kind of observation has been qualified as not automatable as well because the data source was filled someday by other researchers. Moreover, if an additional property is needed, it probably cannot be obtained automatically.

The papers give 16 experiments which can be automated. This is almost 18 percent of the total number of experiments. Furthermore, there seems to be a trend towards more automatable experiments. However, more research is needed to confirm this.

Automated Research  The next aspect of the evaluated experiments was whether or not the experiment is automated. An experiment was evaluated to be automated if the observation, abstraction and analysis is done automatically. So, if the experiment can be performed by giving only one command or one set of commands. If no automation was described, the experiment was evaluated as not automated. The tools that were described in [41] and in [21] are similar to our infrastructure, therefore, these will be described in Chapter 6.

Seven of the experiments in the papers are automated. This is less than half of the experiments that can be automated.

Tool Support  In the previous paragraphs, we have seen that sixteen experiments can be automated, however, only seven are automated. Therefore, the development of a general infrastructure to automate empirical experiments might not only be useful for the Delft university of technology but also for researchers elsewhere. Furthermore, all quantitative experiments can be partly automated, so, the infrastructure can also be used in those cases.
Chapter 4

The Empirical Research Infrastructure for Clustered Software

This chapter will address the need for automated experiments with the description an prototype of the Empirical Research Infrastructure for Clustered Software (Erix)\(^1\).

The description contains the requirements of the infrastructure, the technical design and the implementation details. Finally, a short introduction is given on how to use Erix.

4.1 Requirements

This sections translates the project goal, the initial ideas and the challenges to requirements for the infrastructure. The project goal is not very specific, it asks to setup experiments effectively, in this section, this is also specified.

In addition to the name and a description of each requirement, the reason of the requirement is also specified. The requirements are ordered by priority (descending).

Requirement 1 *The infrastructure should conduct user-defined experiments automatically*

**Description:** This is the most important requirement and specifies the normal use of the infrastructure: the components of an experiment (input, output, experiment steps and experiment model) need to be specified in order to define an experiment. If the experiment is defined, the user can execute the experiment, the data collection and analysis will be executed automatically. **Why:** The reasons for this requirement is to simplify the process of empirical research for the user in order to save effort. This is the main motivation for Erix. A challenge which can be fulfilled by this requirement is the cost reduction challenge. Since the cost will reduce if the setup is easier and parts of the experiments are automated.

\(^1\)Although the acronym of Empirical Research Infrastructure for Clustered Software is actually ERICS, Erix is chosen as the short name for the infrastructure to emphasize the close connection to Nix and to remove the direct link to the name of the author of this thesis.
4.1 Requirements

Requirement 2 The infrastructure must be able to conduct clustered software measurement research.

Description: In Section 3.2 we have defined clustered software measurement research. The user must be able to define and conduct this kind of research using the infrastructure. In such an experiment, the same experiment is conducted on several SUAs. Why: We want to conduct clustered software measurement experiments to gain knowledge from a whole population of products instead of a single product.

Requirement 3 The infrastructure should use standard measurement and analysis software

Description: There exists many ready to use software tools to take software measurements and analyze measurement data. The infrastructure must be able to use these. This must be done by giving shell-commands. Why: Using ready to use measurement tools saves time.

Requirement 4 The (interim) results of the research must be stored

Description: All the results of the experiments conducted with the infrastructure should be stored. This includes the results of the separate research steps (interim results). Some steps have interim results of their own, it is not required to save these as well. Why: To support the reuse of the (interim) results, the results must be stored. The reuse of the results could be useful if the calculation of the results takes much time. Also, the results provide inside into the results of parts of the experiment. Furthermore, the replication challenge demands that the results are saved.

Requirement 5 It must be possible to reuse (parts of) the experiments.

Description: The goal to support replication and aggregation demands the possibility to reuse parts of the experiment. Notice that two types of artifacts might be reused in experiments: the definitions and the (interim) results. The second type of artifacts is already covered in the previous requirement. The definitions that are used must be saved to be used again in other definitions. Why: Reuse of parts of the experiment definition makes it easy to redo the experiment with or without minor changes. This is important to lower the costs of replication and aggregation.

Requirement 6 The infrastructure should be OS independent.

Description: The infrastructure should be run on every used operating system by the users. It is also allowed to install the infrastructure on a server if the server can be operated by the users. Why: The availability of the infrastructure should be high to give many users the advantages of it.

Several initial ideas from the introduction do not occur in the list of requirements. The reason why these are not requirements is listed below.
**Built-in analysis procedures** All the analysis procedures can be put in the experiment steps. Therefore, we decided not to develop an integrated component to do this. However, in the future, an experiment step can be developed to handle most of the data analysis tasks.

**Online and offline experimenting** The offline experimenting can be seen as an important function of the results of the experiment. This function is the same as using a data sheet as an experiment input. Therefore, online and offline experiments can be automated in the same way.

### 4.2 Technical Design

The previous section specified the infrastructure’s requirements. The upcoming section discusses the technical realization of these requirements. Several options on implementation have been investigated, this section describes these options and the choices that were made.

#### 4.2.1 Implementation Platform

Before the start of the implementation of the infrastructure, it must be known how to build the infrastructure. There exist no requirements or restrictions about the implementing platform/language. Thus, several options have been examined during a preliminary inquiry. These can be divided in imperative and declarative solutions.

**An Imperative Solution**

The infrastructure could be built using an ordinary imperative language such as C++ or Java. However, because of the many shell commands that are called in the use of the infrastructure, a script language is a more logical solution. The chosen script language was Ruby. Four components where identified for the infrastructure: (1) the research core, (2) the user interface, (3) the storage and the (4) output component. Figure 4.1 shows the components and the connections.

![Figure 4.1: The components of the imperative solution.](image)

Ruby is an object oriented open source script language. For more information about Ruby, see [http://www.ruby-lang.org/](http://www.ruby-lang.org/).
4.2 Technical Design

**Research core:** The task of the research core is to take care of the communication between the other components. Moreover, the main task of the research core is to conduct the research according to the definition.

**User interface:** The user interface gives the user an appropriate method or tool to define and execute the experiment. When the experiment is done, the user interface returns the results to the user. A preliminary idea was to implement the user interface as a web site.

**Storage:** The task of the storage component is to store all the definitions, interim results and results that need to be stored. This component is mainly responsible for the ability to reuse research definitions and research results. Recall from Section 2.4.3 that all independent, confounding and dependent variables need to be stored. This can be done in a database.

**Output Component:** The output component handles the special output tasks such as generation of data tables and diagrams.

An object model has been created to illustrate the operation of the imperative infrastructure. Figure 4.2 on Page 33 shows the preliminary object model.

In the object model, the research components are represented by the objects. Besides these components some other classes are shown. Instances of the StepConnection class represent the connections between the steps in a research model. The ExperimentInput and ExperimentOutput classes represent the users input and output. This is connected to the model’s input and output.

A final tool that is needed for the imperative solution is a research definition parser. This parser translates the research definition (created by the user) to appropriate models. The ASF+SDF meta-environment can be used for this function [50]. The ASF+SDF meta-environment supports the definition of a language and it generates the needed tools to use the language (e.g. parsing).

**Declarative Solutions**

Declarative is the opposite of imperative. Imperative languages describe how to reach the goal of the software. Declarative languages describe the goal itself. We will concentrate on build systems, software tools used to build software. The building process consists of several steps, dependent on the programming language. For example, the building process of C software consists of a compile step and an link step.

Software is usually built from several files, the amount of files may vary from one to several thousands. A file can depend on other files, this way one or more dependency-trees can be constructed. A single change in one file can cause a rebuild of all the files. Build systems are used to decide if files need to be rebuilt or not. Besides this task, build systems can be used to do adjacent tasks such as the software configuration [27].

The compiling, linking and configuration is done by executing shell commands. For instance, the compiling of a C file is done using the command $gcc <inputfile> -o <outputfile>
Recall from the definition of software measurement research and clustered measurement research that the measurements and analysis must be done by executing shell-commands in a certain order. This means that a build system can be used to conduct software measurement research because: (1) the order of the measurements and the analysis steps can be guaranteed by using the build system dependencies; (2) the measurements can be modeled as the build of the measurement results.

The next paragraphs describe a classic build system, Gmake, and a deployment system, Nix. In addition to a short description, the possibilities to use these systems in the infrastructure will be emphasized.
4.2 Technical Design

The Empirical Research Infrastructure for Clustered Software

Gmake  *Make* was the first build system, it was introduced in 1979 [19]. *Make* is used by writing *makefiles*. In a *makefile* several *rules* are defined. Rules are the building blocks of makefiles, they consists of one or more *targets*, the files that *Make* targets to build, zero or more *prerequisites*, files needed to build the target, and zero or more *commands*, shell commands that use the prerequisites to build the target. If a target needs to be build, *Make* first checks if the prerequisites are up-to-date. If a prerequisite is the target of another rule, this rule is executed, otherwise the time stamp of the prerequisite is compared to the time stamp of the target. If the prerequisite time stamp is more recent than the target time stamp, the commands are executed. This results in an up-to-date target.

It has been mentioned before that *Make* could be used as a research platform (e.g., in [6]). To investigate this possible platform, a small test case has been developed. This test case profiled a software product dynamically. Listing 4.1 shows the makefile, responsible for the building process and the execution of the newly build software.

```
1  include <source_makefile>

CFLAGS += profiling-flags
LDFLAGS += profiling-flags

6  *.gcda : executable, input
       run program for each input
```

Listing 4.1: Pseudo code of a makefile

This example contains one rule, the target is *.gcda and the prerequisites are executable and input. Makefile Rules are static, this means that the dependencies and targets need to be specified by the user. As a result, the experiment defined using *Make* is not very flexible. This makes the *Make* solution very knowledge intensive for the user. The makefiles could be generated but that is actually a more sophisticated version of the imperative solution, because a makefile must be delivered instead of a list of commands.

Nix  *Nix* is a *safe and policy free system for system deployment* [14]. Software deployment is more than just building, it is the step between implementation and use of software components. Nix presents some features which other deployment systems lack (e.g. RPM, RPM Package manager, and dpkg, Debian package management system). The most innovative feature is the possibility for concurrent installations of multiple versions of components. This feature makes it safe to upgrade components because the old version is not overwritten, it can still be used.

The Nix system consists of the following components: the store, user environments and Nix expressions. In a build context, the Nix store stores the deployed software components. the user environments make the components available for the users and Nix expressions are specifications of the construction of components. they drive the building and installation of components.

Every Nix package has a derivation, this is a special function that specifies the variables needed to build the package. In a derivation, the builder must be specified, this is a shell scripts that actually builds the package with the use of the specified variables. A build
package can be found in the Nix profile(s). A profile consists of links to Nix packages in the Nix store.

Another feature of Nix is that it is policy-free. This means that a wide variety of policies can be applied on the Nix elements. An example of a policy is that some users cannot change the store, or use a particular user environment.

Nix expressions are written in the Nix expression language, a pure, lazy functional language [22]. In Appendix C, a short introduction is given of the Nix expression language.

Nix can be used in the infrastructure in the same way as Gmake was used. However, the Nix expression language can be used for a more sophisticated solution: the definition interpreter. This way, a user defined research will be interpreted by an interpreter, written in the Nix expression language. The definition itself needs to be written in Nix as well. This way, a new definition language can be constructed according to the requirements. Note that the definition language must be within the limits of the Nix language, no new constructs, such as loops, may be introduced.

The Nix definition [22, 14] reports that the Nix “building blocks” are packages. However, the experiment definition contains experiments, experiment models, and software commands. In the Nix environment, these components need to be represented in packages.

One of the challenges in empirical research in software engineering is the support of replication and aggregation. It has been identified that the support of these features can be reached by the storage of the research definition, context and results. Using the Nix environment, the storage of the definitions and the results are solved. Since different versions of the same software can be installed concurrently, the results of different runs of the same experiment can be stored as well. The definitions of the experiments need to be saved too, however, the Nix definitions are just text files, thus, this is no problem. This also means that the definitions of experiments can also be reused.

The Nix solution has some serious advantages over the make solution: more sophisticated definition interpreter, built-in storage, easier reuse. Therefore, we compare the imperative solution only to the Nix solution.

### 4.2.2 Choice

In the previous sections, several implementation platforms have been introduced. In this section, these platforms will be compared and a definitive implementation platform will be chosen. The options will be evaluated on the following criteria: Development effort, flexibility / extensibility, simplicity, requirements support and comprehensibility for the research definition.

**Development effort** If the infrastructure will be built using the imperative style, more actions need to be programmed, such as the definition > objects conversion. Moreover we predict some difficulties implementing the imperative solution. During the preliminary inquiry in Nix, we already developed an infrastructure that could conduct a simple research.

**Flexibility / extensibility** Although the requirements are clear for now, these might change someday. The question is if the preliminary designs are capable to change with the require-
ments or is it necessary to revise the infrastructure completely? The imperative infrastructure can be changed and extended freely, as long as the new requirements can be met with a computer, it can be programmed (as long as the used language is Turing complete). Note that big changes in the requirements may need big changes in the infrastructure. If the infrastructure is implemented using Nix, the range of changes which can be applied in is more limited. Some of the changes cannot be applied if we use Nix. For instance, we can add an extra requirement that specifies that an experiment must be run synchronous on two computers. To synchronize the experiments, network communication between the computers is needed. In Nix, network communication is not possible, so this extra requirement cannot be implemented in Nix.

**Simplicity** It is clear that the Nix infrastructure is simpler than the imperative solution, this would contain lots of classes, variables and methods where the Nix solution only has several interpreters.

**Meeting the Requirements** The preliminary experiments to discover the opportunities of both the Nix solution and the imperative solution have shown that the requirements can be met with both options.

**Research definition comprehensibility** The comprehensibility of the definition is important for the usability of the infrastructure because the definition needs to be written by the user. The experiment definition of the imperative style must be defined itself, this can be done in several ways. The definition of the Nix solution must be in Nix-style. The provisional definition has been proposed to the user, and the definition was verified to be comprehensible enough.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Imperative</th>
<th>Nix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development effort</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Flexibility / Extensibility</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Simplicity</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Requirements support</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Comprehensible definitions</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 4.1: Imperative vs. Nix review

In the previous section it is already shown that the requirements can be fulfilled by both solutions. To make a decision, we have to make a decision about the priority in the “secondary criteria”. The conclusion is that the development effort is the most important. Thus, the infrastructure will be build in Nix.
4.2.3 The Design

The choice of Nix as the implementation platform has several implications for the implementation and the technical documents.

Definition Translation

The research definition needs to be translated to a list of commands that must be executed. Recall that the Nix building blocks are packages and that these packages can be built by executing the package’s derivation. This derivation contains a reference to a shell script, the builder, that executes the commands. In the case of a research definition, the real commands that need to be executed are enclosed in the research steps. Thus, for each step, there should be a Nix package to execute the step’s command(s). These packages can be seen as a third layer in the experiment model of Figure 3.1 on Page 26, as shown in Figure 4.3. We call this the software-run layer.

Figure 4.3: Model of a software measurement research including the Nix packages
The fourth layer of is called the software layer; all kinds of software can be used within the packages of the experiment package layer. The used software packages are also dependencies of the experiment. Therefore, these dependencies can be enforced by adding a dependency to the used software package. It is not essential that the software is added to the software layer. Arbitrary commands can also be given in the software run packages, however, the software is not controlled in that case. So, it cannot be guaranteed that the software exists.

Storage Design

We have seen several requirements that need the storage of certain data. To consult the results of an experiment after it is done, the results and interim results must be saved. Furthermore, the experiment definition must be stored in order to reuse it.

The choice for Nix as the implementation platform implicates that all experiment results will be saved in the Nix store. All the results are accessible from the Nix default profile. This makes the implementation of the experiment results storage very easy because Nix takes care of this.

To store the definitions in a logical way we introduce a directory structure (Listing 4.2). All definitions are stored in one of these directories or in one of their subdirectories. Adding and removing definitions from this definition repository is done with special tools that will be introduced in Section 4.3.3.

| - Models  
| - Experiments  
| - Software  |

Listing 4.2: Erix definition repository directory structure.

Experiment Definition Design

The components that need to be defined have been described in Section 3.3. This section specifies how these components can be defined with respect to the infrastructure requirements.

Three issues have been identified on the design of the definition language: (1) the need for reuse of the experiment model definitions instructs to specify the experiment-specific variables in the experiment definition; (2) the need for clustered software measurement experiments implicates the need for some looping mechanism to conduct the same experiment on different SUAs; (3) it would be useful to allow steps to contain references to experiment models. This way experiment models can be used within other models. This improves the opportunities for reuse of models.

The first issue can be explained using Figure 4.3 on Page 37. We state that the model layer, the software run layer and the software layer may not contain any values that are specific for that experiment. This is legitimate because if any of these layers contain data from a particular experiment, it can only be used in that experiment. All the variables in the lowest three layers that depend on the input of the experiment, should get the variables from
the layer directly above. For instance, the execution of a piece of software (this takes place in the software run layer) needs a parameter, this parameter needs to be specified in the experiment layer. The parameter is passed on to the software run layer via the experiment model layer. This implicates that a definition needs some variables to be filled in by the packages in the layer above.

The second issue orders to implement a looping mechanism. The goal of this looping mechanism is to repeatedly execute an experiment step on different input. So, the experiment definition must provide more than one set of variables for the execution of the model.

An additional feature of the experiment definition is to allow the use of references to experiment models within experiment steps. This allows the user to execute an experiment model within an other experiment model. The models can be reused more often because more general models can be defined. Together with the looping mechanism, this becomes a powerful tool to create clustered software measurement experiments. This can be done with two models, the first defines the experiment steps for only one SUA. The second model loops over the input and refers to the first model.

4.3 Implementation

This section discusses the implementation process of the infrastructure prototype and the result: the prototype itself. This is done by describing several scenarios and solutions for these scenarios.

4.3.1 Research Definition Interpreters

The choice was made to use interpreters for executing the results. The interpreters responsibility is to build the packages according to the research definition. Five different types of experiment packages have been identified: experiment, experiment-model, experiment-step, software-run and software packages.

**Experiment interpreter** The experiment interpreter interprets the experiment definition. Recall from Section 3.3 that an experiment should contain a model and is supplied with appropriate input. The interpreter’s goal is to run the specified model with the input. In Listing 4.3 the experiment interpreter is shown.

```plaintext
experimentInterpreter = experiment :
  pkgs.stdenv.mkDerivation {
    inherit (experiment) name;

    builder = ./experimentbuilder.sh;
    saveReplications =
      if experiment ? saveReplications then
        experiment.saveReplications
      else
        "false"
    ;
```

39
4.3 Implementation

The Empirical Research Infrastructure for Clustered Software

```
buildInputs =
  if builtins.isList experiment.inputs then
    map {input: modelInterpreter input} experiment.inputs
  else
    modelInterpreter experiment.inputs
 ;
```

Listing 4.3: The experiment interpreter

Line 1 indicates that this interpreter is a function with one parameter: the experiment definition to interpret. All data of this definition is passed to the builder and the model interpreter. The builder copies all output of the execution of the models to the appropriate directory. The attribute `saveReplications` indicates if the replications of the experiment must be saved explicitly. This can be useful if the experiment is executed once in a while and the differences of the runs is important. If `saveReplications` is set to false, then the former experiment results are not accessible with the default (Nix) profile. However, these results can be found in older profiles.

**Model interpreter** A model definition is interpreted by the model interpreter. The model definition contains the model’s name and the steps. The interpreter checks if the model contains any steps. See Listing 4.4 for the model interpreter.

```
modelInterpreter = package :
  pkgs.stdenv.mkDerivation {
    inherit (package) name;
    builder = ./modelbuilder.sh;
    buildInputs =
      if package ? steps then
        map {fstep: stepInterpreter (fstep stepInterpreter)} package.steps
      else
        []
    ;
  };
```

Listing 4.4: The model interpreter

The model interpreter is a function as well, it accepts one parameter that is called `package`, of the type model definition. The attributes in `package` are passed on to the builder and the `stepInterpreter`. Line 6 and 7 checks if `package` contains steps, and if so, each step is executed using the step interpreter.

Line 4 contains the specification of the builder. The purpose of the builder is to copy the output of the steps to the output directory of the model.

**Step interpreter** The third interpreter is used to interpret the step definition. The step definition is a part of the model definition, it defines the attributes for a single experiment step. In Section 4.2.3 we identified that the steps can be executed iteratively and a model can be specified as a command in a step. This has consequences for the interpreter.
The Empirical Research Infrastructure for Clustered Software 4.3 Implementation

```haskell
stepInterpreter = step :
  # Check if the step has a loop
  if step ? loops then
    map (loop : stepInterpreter loop) step.loops
  else
    # Check if the step is actually a model
    if (step.prog step.inputs) ? steps then
      modelInterpreter (step.prog step.inputs)
    else
      # Check if the used software run has its own builder
      if (step.prog step.inputs) ? specialbuild then
        (step.prog step.inputs).specialbuild
      else
        # The standard derivation.
        pkgs.stdenv.mkDerivation rec{
          name = (step.prog step.inputs).name;
          inherit (step.inputs) src;
          command =
            if (step.prog step.inputs) ? builder then
              "$**$
            else
              (step.prog step.inputs).command;
        builder =
          if (step.prog step.inputs) ? builder then
            (step.prog step.inputs).builder
          else
            ./builder.sh;
        }
```

Listing 4.5: The step interpreter

Every step definition contains the attribute `prog`, this is the definition of the software-run package. With the expression `step.prog step.inputs` the appropriate software-run definition is identified.

The step interpreter in Listing 4.5 contains five if statements. The first if (line 3) statement checks if the steps contains any loops, if it does, the step interpreter is called for every loop. The second if statement (line 7) checks if the input `step` is really a step or a model. Recall from Section 4.2.3 that a step can refer to a model. If a model is identified, the model interpreter handles the definition. The third if statement (line 11) checks if an attribute `specialbuild` is defined. If `specialbuild` is not defined, a default builder is used to execute the step. Unfortunately, the default builder is only suitable for very simple steps. If `specialbuild` is defined, the derivation is defined in the step definition itself.

The software-run and software packages does not have separate interpreters. These are fully defined Nix-packages and have their own derivations and builders.
4.3 Implementation

4.3.2 The Infrastructure Directories

We have seen several types of packages and corresponding definitions. To prevent confusion of the definition files, this section proposes a directory structure for the definitions. This directory structure is shown in Listing 4.6 on Page 42.

```
- Rootdirectory
  - Models
    models.nix
    <model-files>
  - Experiments
    experiments.nix
    <experiment-files>
  - Software
    - <software dir>
      <software>.nix
      builder.sh
      software.nix
      infrastructure.nix
      builder.sh
      experimentbuilder.sh
      modelbuilder.sh
```

Listing 4.6: The infrastructure directory structure.

The models directory and the experiments directory both contain one file (models.nix and experiments.nix) containing a list of available package definitions. The software directory also contains a file that lists the available software run definitions. However, the full definitions are not in the software directory but in a sub-directory, because the software packages may contain their own builders.

The root directory\(^3\) contains 3 builders: experimentbuilder.nix and modelbuilder.nix for executing the experiments and models respectively and the builder.nix. Builder.nix is the software run builder if the software run definition does not specify a builder. The last file in the root directory is infrastructure.nix, this file contains the interpreters.

4.3.3 Infrastructure Tools

In this section, the tools that represent the infrastructure for the user are presented. The commands that are needed to use the infrastructure are introduced and a explanation of their use is given.

Search Tools

**Command 1** `searchdef <type> <searchstring>`

The searchdef tool search for the given searchstring in the definitions of the given type. Type can have 3 values: `experiment`, `model` and `software`. The searchstring parameter can

---

\(^3\)The name root directory is arbitrary here, the root directory can be any directory and does not intrinsically refer to the “/” directory (in *nix).
be a regular expression using the grep syntax. The tool returns the definitions of the given that contain the search string.

So, the command searchdef model "test" returns all model definitions which contain the word test. The filename and the definition is printed on the screen.

**Command 2** `searchresult <name>`

The searchresult tool searches for the specified name in the experiment results. The tool returns a list of paths of the selected packages in your personal Nix profile.

**Definition Tools**

**Command 3** `adddef <definition-filename>`

This command inserts the definitions from the specified file into the infrastructure definition files. The tools checks if the defined names in the new definition file do not conflict with the existent definitions. If the names conflict, the user can still choose to add the new definitions. The result of the command is that the definitions are added to the infrastructure’s definition files.

**Command 4** `testdef <definition-filename>`

The testdef tool executes three steps, first, it adds the definitions in the specified file to the infrastructure’s definition files. Second, the tool will run the experiment using the nix command `nix-build` that executes the experiment but it will not put any symlinks in the user environment. Subsequently, the definitions are removed from the infrastructure’s definition files to recover the original state.

**Remaining Tools**

**Command 5** `runexperiment <name>`

The runexperiment command runs the specified experiment. The name that is needed is the name of the experiment in the definition. If the experiment is added by one of the definition tools, the name is the filename without the extension.

**Command 6** `gotoresult`

This command makes the profile directory the current directory.

---


5[^5]: If the name is found within a file, the adddef tool will point out that there might be introduced double names. This is not entirely sure because the found name can be found within a definition, in that case, the addition of the new definition will not introduce a conflict. Therefore, the user can add the definitions anyway.
4.4 How to Use Erix

This section describes how the infrastructure prototype works, and why. This section gives a short introduction on the use of the infrastructure.

4.4.1 The Installation of Erix

Before Erix can be used, it must be installed. To do this, the following steps need to be followed. Note that Nix must be installed.

- Download the tarball.\(^6\)
- Extract the tarball to an arbitrary directory.
- Create a symlink to the `pkgs` directory in Nix packages, this link must be named `pkgs`.
- Erix can now be installed using the command `nix-env -f default.nix -i Erix-1.0`.
- Find the Erix package in the Nix store (erix directory in the most recent profile).
- Edit the `definition_root` variable in `config.py` to the `/nix/store/...` directory.
- Again, create a `pkgs` symlink to the Nix-packages `pkgs` directory in the erix directory.
- The rights of the erix directory must be set to give all users read and write access. This can be done by executing the command `chmod -R +w erix`.

Now, Erix is fully operational and can be used as described in the upcoming sections.

4.4.2 A Research Definition Example

This section introduces a very simple example to give a feeling of the definitions. The goal of this experiment is to write the directory listing in a file (`example.txt`). This file is the output of the research.

Research Definition

The research definition in Listing 4.7 consists of two parts: the list part and the actual definition. The list part is copied to the file `experiments.nix` and specifies where the definition can be found. The actual definition, `example\_exp.nix`, defines the needed attributes.

```
experiments.nix:
    example_experiment = (import ./example_exp.nix) {
        inherit (models) example_model;
    };
```

\(^6\)http://swerl.tudelft.nl/bin/view/Main/ErixInfrastructure
4.4 How to Use Erix

The definition in experiments.nix contains the line `inherit (models)example\_model;`. This is needed because the definition `example\_model` is not known within the definition. This way the model definition is passed to the experiment definition. This experiment contains two attributes: `name` and `inputs`. The `name` attribute is obligatory. The `inputs` attribute contains a link to the experiment model, `example\_model`, with the input for the model.

**Model Definition**

The model definition is also split up in two parts: a list part and the actual definition.

```nix
example_exp.nix:
{ example_model }:
rec {
    name = "example_experiment-0.1";
    inputs =
        example_model {
            src = /home/af06011/experiment_input/input/example ;
        };
}
```

Listing 4.7: The experiment definition

```nix
models.nix:
example_model = inputs: {import ./example.nix} {
    inherit (software) example_run;
    inherit inputs;
};
example.nix:
{ example_run, inputs }:
rec{
    name = "Example-0.1";
    steps =
        let
            step1 = interpreter :
                {prog = example_run;
                 inputs = {
                    inherit (inputs) src;
                 };
             };
        in
            [step1];
}
```

Listing 4.8: The model definition
The model also contains two attributes. Again the \texttt{name} attribute which is obligatory for every Nix Package.\footnote{Although this is not really a Nix package, the interpreter creates an Nix package from this attribute set. Therefore every attribute set that is interpreted by one of the Erix interpreters must have an attribute \texttt{name}.} The second attribute is \texttt{steps}. \texttt{steps} contains the steps of the experiment. It might be remarkable that the \texttt{step1} function has a parameter named \texttt{interpreter}. Each step needs this parameter because the interpreter is required to execute the dependencies if there are any. The \texttt{prog} attribute indicates the software run package that must be built to actually execute the step. All input variables that the program run package specified at \texttt{prog} need to be in the input attribute set.

**Software Run Definition**

A software run definition can be divided in three parts: The list part, the definition and the builder.

```
software.nix:
  example_run = inputs: (import ./example/example_run.nix) {
    inherit (pkgs) stenos;
    inherit inputs;
  };

./example/example_run.nix:
{ stdenv, inputs } :
  rec {
    name = 'example_run-0.1';
    specialbuild = stdenv.mkDerivation {
      inherit name;
      inherit (inputs) src;
      builder = ./builder.sh;
    };
  }

./example/builder.sh:
  source $stdenv/setup
  ensureDir $out
  cp -r $src/* ./
  dir > example.txt
  cp example.txt $out
```

Listing 4.9: The software run definition

In the list-part the \texttt{stdenv} is inherited, this package contains all kinds of standard shell commands such as \texttt{cd} and \texttt{dir}. Moreover, it contains the \texttt{mkDerivation} function which we use in \texttt{example\_run.nix}. In this derivation, we put the attributes needed by the builder. The
The Empirical Research Infrastructure for Clustered Software 4.4 How to Use Erix

The builder itself is very simple, it copies the files of the src directory to the current directory and it stores the output of the dir command to the file example.txt. Finally, the example.txt file is copied to the output directory (the output directory is indicated with the \$out variable).

4.4.3 The Definition Template

To make it easy to add definitions to the definition repository, all research definitions can be defined in one file, this definition file is split up into the files mentioned above. The definition file must have a certain format, this format is discussed in this section.

The experiment definition is a combination of all files and file-fractions that specify an experiment. In order to split these definitions, a new “language” has been specified. This language defines the partitions and some additional syntax. Together with this specification language, an experiment definition contains three different syntax: the Nix syntax, the bash syntax (builder) and the specification language syntax.

Within the definition files are a lot of connections. Three types of connections have been identified: file connections, attribute connections and count connections. The file connections are references to other files in the same definition. To enforce the connection, the name should be the same. Attribute connections are attributes that are passed to other functions or files, it is important that all attributes that are expected are passed. Count connections have been introduced by the specification language. To parse the definition file correctly, we need to know how many definitions are specified. This is indicated by a number. Listing 4.10 shows the definition template.

```nix
### experiments.nix ###

count=<Z>
<experiment name> = (import <experiment definition file>) {
  ...
  #enddefinition

### models.nix ###

count=<Y>
<model name> = inputs: (import <model definition file>) {
  ...
  #enddefinition

### software.nix ###

count=<X>
<software name> = inputs: (import <software file>) {
  ...
};
#enddefinition

### experiment ###

count=<Z>
```
4.4 How to Use Erix

The Empirical Research Infrastructure for Clustered Software

```plaintext
experimentfilename=<experiment definition file name>
{
  <parameters> 
}

rec 
  name = '<name>'; 
  inputs = <model> [ 
    ... 
  ];
#enddefinition

### model ###
count=<Y>

modelfilename=<model definition file> ...
#enddefinition

### software ###
count=<X>

softwarefilename=<software file> ...
#enddefinition
```

Listing 4.10: The research definition template.

4.4.4 The Output Directories

Just like a structure of the definition files and infrastructure files, the output is also structured in a specific manner. This section shortly clarifies this structure. Listing 4.11 shows the basic structure of the output.

```plaintext
/Nix/store/<hash>-experiment-name/
   <model-name> (symlink)
       step1-<stepname> (symlink)
       step2-...
```

Listing 4.11: The infrastructure output directory structure.

The experiment directory contains a symlink to the model directory with the name of the model. The model directory contains symlinks to the directories of the different steps, named step$<number>\_$<stepname>. The number is just a counter to indicate in which order the steps were executed. Recall that a model can be used as a step, this is represented in the output as a step which contains steps of its own.
An extra aspect can occur in the output directories: iterations. If the model contains a loop, the output must reflect this. If a loop is encountered (the first step name is observed more than once) a directory is created with the name iteration1. For each iteration, a new iteration directory is created.

4.4.5 Research Scenarios

This section describes several scenarios to illustrate how to use the infrastructure prototype in these particular situations.

Scenario 1 Define a new research and conduct it.

To conduct an experiment, the definition must be added to the infrastructure’s definitions. This definition can be created using the template specified earlier. If the definitions have been specified, these need to be added to the infrastructure’s definitions. This can be done in two ways, the first is to call the command adddef which simply adds the definitions to the right files. The user must then run the runexperiment tool to execute the experiment. The second method is to call deftest, this script adds the definition to the right files and then conducts the specified experiment. Afterwards the definitions are removed from the definition repository.

Scenario 2 The user wants to specify a new experiment, based on an existing model. Where to find the necessary parameters?

The necessary parameters can be found in the model definition. To find this definition, the searchdef tool can be used.

Scenario 3 Replicate an experiment

The description of replication in Section 3.1 reveals that that several inputs can be variable if an experiment is replicated. This can be done by creating a new experiment and using the same model. It is probably the easiest way to start with the definition of the original experiment and modify the input variables and the name of the experiment. The new experiment definition can be inserted in the template and added to the infrastructure’s definition in the usual way.

If the replication is done with the exact same input, the infrastructure must be forced to redo the experiment because Nix will notice that the inputs did not change. Erix can be forced to redo an experiment by renaming the experiment definition.

Scenario 4 Aggregate an experiment

An aggregation of an experiment requires a more rigorous modification than an replication according to the definition of Section 3.1. In the case of an Erix experiment, this means that not only the experiment definition is modified but also the other components such as the model. The definition can be created using the same method as described in Scenario 3: copy the model and experiment and modify the needed variables.
Scenario 5  An experiment has already been conducted, the user wants to search for the results.

Two methods can be chosen: The first is to go to the profile directory and look for the experiment name in the experiment directory. The second option is to use the searchresult tool. This way, all the available experiments with the given string are found.

Scenario 6  Export the experiment

Currently, the infrastructure prototype does not support the exportation of complete experiment definitions (including the necessary model and software definitions). This can be done manually by copying the separate definitions one by one to a definitions file.

4.4.6 Errors

During the testing of an experiment definition several errors may occur. This section gives some of the most encountered errors during testing.

Looping errors  Using the looping option, infinite loops may occur, Nix will encounter this and gives the following error message: infinite recursion encountered along with the line-number and filename where the error was encountered.

Missing attributes  Attributes are passed by the various packages to other packages. It is important that all packages that are needed are present. The error message that is shown if an function expects an attribute that is not passed: attribute ‘<attribute>’ missing. The converse does not give an error, so if an attribute is passed which is not used.

Note that an attribute that is in all four layers of the model (shown in Figure 4.3 on Page 37) should be in four definitions: (1) the experiment definition; (2) the model definition (in the steps); (3) the software definition; and (4) in the software builder.
Chapter 5

Erix Experience in a Case Study

This chapter gives a description of a case study. This case study is used to illustrate the use of Erix. The case is taken from [6], and has already been described in Example 2.1 on Page 9. This case has been chosen because it is well known within the SWERL department. As a result, we can concentrate at the conversion of the case study to Erix instead of the experiment itself.

Recall that the goal of the case study is to test the infrastructure’s functionality not to use the results of the case studies in a research. This means that the experiment results are not analyzed further. Instead we focus on the evaluation of creating a new research using Erix.

5.1 The Inspection Results Prioritizing Experiment

The goal of this experiment is to test if it is reasonable to use statically determined execution likelihood instead of dynamically determined execution likelihood for prioritizing software inspection results.

The experiment is conducted by executing several commands. These commands use Codesurfer\(^1\) (csurf) and perl\(^2\) with appropriate scripts to get and analyze the software measures. Besides these, GCC and gcov\(^3\) are needed to dynamically profile the SUAs.

5.1.1 The Experiment Steps

The experiment consists of five steps: (1a) codesurfer compile, (1b) static execution likelihood analysis, (2a) gcov copy (2b) SUA instrumentation and profile extraction (3) comparison of dynamic and static execution likelihoods. For each of these steps one or more commands need to be executed. These commands together with the supposed output is listed below.

\(^1\)www.grammatech.com/products/codesurfer/
\(^2\)http://www.perl.org
\(^3\)http://gcc.gnu.org
5.1 The Inspection Results Prioritizing Experiment

1a Codesurfer builds the SUA. The command is: `csurf hook-build <project name> - preset-build-options highest --- make <target>`. The result of this step is the build project.

1b Codesurfer determines the execution likelihood statically. This is done by running a codesurfer script: `csurf -nogui -l <script location>/bb.stk <project name> <project name>.pred`. This step delivers a .pred file.

2a gcov copy. Gcov is not standard installed in Nix and it is difficult to build within Nix because it is a part of the GCC compiler. Therefore we build the correct version of gcov outside Nix. To use it, gcov must be put to the Nix store, this is done in this step.

2b Build the SUA with profiling flags and run it. This is done by executing two commands:

```
make CFLAGS=''-fprofile-arcs -ftestcoverage'' LDFLAGS=''-fprofile-arcs'' all
```

It is assumed that Gmake is used and that all is the build target. The second command is to run the SUA with specified input. In order to get a proper representation of the normal use, the SUA is run with several inputs. The result of this step is a series of .gcda files out of which the profiles can be extracted. Gcov can be used to extract the SUA's profile. All .gcda files need to be considered. The command to do this is `gcov -b <.c file>`.

3 Perform statistic analysis on the gathered data to deliver the appropriate results. The analysis is executed by a perl-script: `perl <script location>/stat.pl antiword.pred <output directory step 2b> <project name>.csv <project name>.stats`. The <project name>.csv and <project name>.stats is the output of this step.

5.1.2 The Experiment Definition

To show how the experiment is defined some fractions of the definition file will be shown in this section.

Model Definition

The first part is the model definition, parts are shown in Listing 5.1 and Listing 5.2 on Page 53. Recall that the model defines which software must be called in which order.

```python
rec{
    name = "Static_vs_Dynamic-0.1";

    steps =
      let
        step1a = interpreter : {
          prog = csurf_build;
          inputs = {
            inherit (inputs) buildcommand src configurecommand maketarget;
            compiler = pkgs.gcc32;
          };
        }
    in
```
5.1 The Inspection Results Prioritizing Experiment

Listing 5.1: The begin of the experiment model of the execution likelihood prioritizing experiment.

```plaintext
projectname = inputs.runcommand;

step1b = interpreter : {
    prog = csurf_script;
}

Listing 5.2: The end of the experiment model of the execution likelihood prioritizing experiment.

```plaintext
inputs = {
    inherit (inputs) scriptdir;
    inherit (pkgs) perl;
    dynamic = interpreter (step2b interpreter);
    static = interpreter (step1b interpreter);
    projectname = inputs.runcommand;
}

in
[step1a step1b step2a step2b step3];

#enddefinition
```

Some remarks can be made to clarify the definition. First, the numbering of the steps is odd, normal numbering (1, 2, 3, etc.) is probably expected. The choice of the names is free. In this situation, this numbering (1a, 1b, 2a, 2b, 3) is chosen to emphasize the dependencies, the b-numbers depend on the a-numbers. Step 3 depends on both b-numbers. Note that the steps will be named step 1, 2, 3, etc. in the output. This is independent of the names in the definition.

A second remark is about the pkgs attribute, this attribute is the link to the Nix packages definitions. The standard software packages definitions can be used with the pkgs attribute.

The third remark is about the syntax of the dependencies. step3 has two dependencies, the both b-numbers. The syntax to indicate a dependency is interpreter (stepX interpreter). What is done in this line is the following, we call the interpreter with the parameter stepX interpreter. What is needed in this line is the output of stepX, unfortunately, the output is not known before it is interpreted. Therefore, the command is given to interpret the step. Because each step has the parameter interpreter, this must be given (the second interpreter in the line).

The fourth remark is about the line [step1a step1b step2a step2b step3]. This list holds the steps of the model that must be executed. Note that this list does not indicate the dependencies.

The model in Listing 5.1 on Page 52 shows two of the steps which have to be executed to get the results of one single SUA. To investigate more than one SUA, we have defined another model, shown in Listing 5.3 on Page 54. The MTO addition in the name stands for more than one.
5.1 The Inspection Results Prioritizing Experiment

Erix Experience in a Case Study

Listing 5.3: The experiment model to conduct the experiment on more SUAs in one run.

In this definition of static_vs_dynamic_MTO we see only one step which contains a loop to iteratively process all its input. The list of inputs for the static_vs_dynamic model is split up and fed one by one.

Experiment Definition

The experiment definition is shown in Listing 5.4. The most pregnant information in the experiment definition is the input of the experiment. All information that is needed somewhere in the experiment is concentrated in the experiment definition.
Software Definition

Finally the software must be run in order to get the experiment results. The prioritizing case study definition contains five software packages. An example of one software package is given in Listing 5.5, note that this definition contains two files.
The first file, `csurf/csurf_build.nix` is the Nix expression, this gives the packages its name and passes the appropriate attributes on to the builder.

The second file is the builder, this shell script contains the shell commands that are needed to generate the output in this case. The given attributes are needed to give the correct parameters and decisions if the configure script must be executed.

In the definitions can be seen that all three levels can contain paths: the gcov location in the model definition, the codesurfer in the software definition and several location in the experiment definition. To support the reuse of the models and software, the environment specific information should be concentrated. The most straightforward location to specify this information is in the experiment definition.

### 5.1.3 Case Study Results

The execution of the experiment did work. Unfortunately, the experiment took too long to be instantly better than all other possibilities. The main cause for this time span problem was probably the need for some experience with Nix and the original experiment which was not designed for Erix. To be sure of these reasons some more research to the use of Erix is needed.

To show that the case study works, the infrastructure requirements are listed below with the comments on the fulfillment.

- A **user-defined-experiment must be conducted automatically** The prototype did work. So, it is clear that this requirement has been met in the case study.

- The **infrastructure should use standard measurement and analysis software** the used software in the case study: codesurfer, perl, gcc and gcov, is standard software, no special features are needed to run them.
• **The (interim) results of the research must be stored** All interim and end results are stored in the Nix-store, the results can be found in the experiment directory in the most recent Nix profile. The directory structure of the output makes it easy to find the intermediate results. For instance the output of the case study has the structure as in Listing 5.1.3 on Page 57.

```
- Nix profile dir (default: `~/nix-profile/
  - experiment
    - static_vs_dynamic_MTO
      - iteration 1
        - step1_Static_vs_Dynamic
        - step2_csrf_script
        - step3_gcov
        - step4_compile_run
      - iteration 2
        ... (same steps as iteration1)
```

• **It must be possible to reuse the defined experiments, models and steps** Two different levels of reuse can be perceived: the reuse of the text of the packages (e.g. copy paste) the reuse of complete packages, a new experiment can use a existing model.

• **The infrastructure must be able to conduct clustered software measurement research** Clustered software measurement is the execution of several steps on more than one SUA. This can be done by adding a loop to the step as in Listing 5.3 on Page 54.

• **The infrastructure should be OS independent** The infrastructure itself is not OS independent, the use of Nix makes it impossible to make it independent of the operating system as long as Nix is not. However, by the use of remote computing (e.g. OpenSSH ⁴ and Putty ⁵) the infrastructure can be used from the majority of operating systems.

Several disadvantages of the use of Erix have been identified:

• The use of the infrastructure costs much time. Initially all the packages need to be defined to conduct a research. However, by this subsequent additions, a repository of definitions is being created. Thus, by the use of the infrastructure, the costs of conducting research will become lower. The initial costs can be seen as an investment.

• To use Erix, two languages are to be known: the bash shell script and the Nix expression language.

The useful lessons of the use of Erix are depicted in the next section.

---

⁵ [http://www.chiark.greenend.org.uk/~sgtatham/putty/](http://www.chiark.greenend.org.uk/~sgtatham/putty/)
5.2 Lessons Learned

The case studies gave a lot of insight of the operation of Erix. This section gives an overview of the encountered issues.

5.2.1 Definition Debugging

Just like any programming activity, the research experiment is not done well at once. The definition probably needs to be “debugged” before it runs without errors. Two types of bugs can be distinguished: syntactic and semantic errors. The syntactic errors comprise the missing “;”, etc. These are the simple problems, Nix will discover them and roughly tell the user where the error is (Section 4.4.6).

The second type of errors, the semantic errors, are errors which prevent Erix to produce the right output. Although every level of the experiment can hold semantic errors, the most will be encountered in the software run layer. The execution of the software is a somewhat delicate process, the call must be perfect to get the right output. A semantic error can be discovered in two ways: The first is the easiest, if a software call is not right, the software can give an error code. The execution of the experiment will be terminated directly. This points the user directly to the software where the error manifests. The second way to discover a semantic error is an incorrect output or no output at all. Some software returns the code that it ended successfully even when an error has occurred (0 code). That way, Nix can not discover something went wrong and continues with the next step in the experiment. Probably the next step cannot run properly because some input is not correct, or the user will discover the wrong output of the experiment. To find this kind of errors the user must check if the output of every step is consistent with the expected output. Therefore the experiment and the separate steps must be well known to conduct it.

If the execution of an experiment has been terminated, no symlinks are created to the results of the experiment steps. This implies that the results cannot be found (except for the Nix store). To check the output of the previous steps, the output is needed. To get the output, the model can be altered in such a way that in the “steps list” only the names of the steps occur which run successfully (in the prioritizing example we can replace [step1a step1b step2a step2b step3] by [step1a step1b] for instance). This way, the experiment runs well and creates the symlinks to the output of every successful step. This output can now be checked and, if these are fine, the next step can be executed manually to find out what the exact error is.

5.2.2 Software Environments

Software runs in some kind of environment. This environment holds all kinds of variables, other software and devices that are required by other software. An example is the license key variable LM_LICENSE_FILE in the prioritizing case study definition. If software is executed, the required environment needs to be set up beforehand.

This issue has been encountered during the case study. GCC 3.2.3 was needed to compile the sources because the measurement scripts needed the gcov 3.2.3 output and versions
of the compiler and gcov are not interchangeable. To make GCC work, the environment is set up by a wrapper function. Unfortunately for the case study (and other profiling experiments) gcov is not copied to the GCC bin directory by this wrapper function. Therefore the case study contains the odd gcov step, which copies a compiled gcov executable to the Nix store.

Although this is actually a Nix issue, this is mentioned here because this can be encountered in the use of Erix.

### 5.2.3 Disk Usage of Experiments

During the execution of an experiment, the output of the experiments are saved on storage device, usually a hard disk. This can become filled up with all kinds of data. The user of Erix should be aware of the size of the experiments, which can be substantial. For instance the output of the prioritizing case study experiment is over 400 MB. During debugging parts of the experiment are conducted over and over again, therefore several gigabytes are filled up in no time.

The user can influence the disk usage considerably, in the software packages of the prioritizing case study definition, the output of the previous steps is copied to the current step. Although it is important to save the original input of the step, this can be done at lower disk usage by linking to the previous steps. Even this can be omitted because the linking is done by the model. If the input is necessary it can be copied to the temporary directory and copy the output file to the output directory.

If the disk is full, the execution of the experiment will be stopped. The removal of unnecessary Nix-package is done by the command `nix-env --gc`.

### 5.2.4 Division of Commands Between Scripts

In the prioritizing case study experiment, several perl scripts have been used for taking measurements and measurement analysis. These scripts can contain actions which could also be handled in the builder scripts of the matching software packages. The division of commands between the builders and the external scripts can be difficult decisions. The experiment definition should be independent of the executing machine, this requirement has consequences for the choice above: (1) paths, filenames and actions that coordinate the inputs and the outputs should be as high in the definition as possible, preferably in the experiment layer (Figure 4.3 on Page 37). (2) The other actions should be placed as low as possible: preferably in the software layer.

Paths and filenames are highly dependent on the machine on which the scripts are executed. An example is shown in Listing 5.5 on Page 55, Codesurfer is called with the line `/home/af06011/apps/codesurfer-1.9p6/bin/csurf`. It is clear that this will only work on the machine where it was developed. It is preferable to concentrate the machine dependent information in one place: the experiment definition. In the case of a machine change, only the experiment definitions need to be modified.

The actions that perform actual measurement and analysis, should be placed preferably in the software-run layer, this makes them independent of the computer where it is executed.
The scripts should have several parameters to control the input, and output so no specific 
directory structure is required. This guarantees that the script can be executed on every 
machine.
Chapter 6

Related Work

This chapter gives an overview of the solutions to similar problems that were described in this thesis. In the first section, other automated empirical tools will be described. The second section discusses other fields in software engineering which expose similar techniques.

6.1 Similar Infrastructures

Several infrastructures have been developed over the years for supporting researchers in empirical experiments. This section discusses several of these. The similarities and differences between the infrastructures and Erix will be discussed.

6.1.1 Squatter Toolset

Offen and Jeffery describe a toolset for conducting measurements in a software measurements program: the Squatter toolset [39]. This toolset supports measurements-oriented-meta-models (e.g. QIP [26], GQM [4]) and captures management and measurement data. The architecture of the squatter toolset is shown in Figure 6.1 on Page 62.

From the figure, it becomes clear that the Squatter toolset uses the S-plus statistics program to deliver statistical results. Another notable detail is the repository which stores the models, raw data and the statistical results.

The Squatter functionality: building empirical models of measurable attributes and entities, perform data analysis and store, interpret and present the results, is similar to the functionality of Erix. A logical step would be to do some research to this toolset and may be even use this toolset in our environment. Unfortunately, [39] is the only paper which describes or mentions the toolset, no more information can be found. Moreover, the toolset itself is not available.

6.1.2 The Emerald Tool

Hudepohl et al. [30] introduced a software metrics tool: Emerald, Enhanced Measurement for Early Risk Assessment. Emerald has been developed to support the quest for more reliability in software, especially for telecommunications software.
6.1 Similar Infrastructures

The source code metrics are gathered by the Datrix [36] tool. Thresholds have been determined for the Datrix metrics, these are used to calculate an out-range metric which indicates the overall quality. Emerald encapsulates some models which are used to translate the metric values into decision making values.

The source code is gathered from a source code library. Furthermore, Emerald is equipped with a graphical user interface which is used to control the measurements and present the results.

The main difference between Erix and Emerald is the lack of flexibility of Emerald. Datrix is used as the metrics tool, the source code is saved in a library and the output is passed on to the user interface. In fact, the Emerald tool could be implemented using Erix, except for the user interface, with one experiment model and several experiments (for every software component for instance).

6.1.3 The GQM Tool

Lavazza describes a tool that partially automates the GQM process [35]. Several metric tools are used to gather the measurements. The outputs of these metric tools are converted to a similar format in XML. This can be used to save all the metrics output in one database.

The use of the GQM tool improves the repeatability of measurements and it makes measurements cheaper.
The differences between Erix and the GQM are: (1) the kind of research of GQM cannot be changed; (2) the GQM results must be transformed to the standard XML format; (3) Erix embraces the whole process of data collection, the GQM tool only uses the results, it does not give the commands.

### 6.1.4 Software Quality Improvement Tool

Tian, Troster and Palma have investigated tool support in software measurement, analysis and improvement [47]. The result of this research is a description of the process and the implementation of a tool. This section concentrates on the tool that has been developed.

A toolset has been proposed to fulfill three kinds of activities: measurement/data gathering, analysis and presentation. The tools and the connections between them are presented in Figure 6.2.

---

**Figure 6.2:** The toolset and its connections (from [47], page 176)

In this figure it can be seen that the S-plus package with S programs is the central component of the toolset. Therefore, the S-plus software and the S programs are used to integrate the other tools.

The advantages of Erix over the toolset are (1) the flexibility and (2) the storage facilities. Erix does not fix the tools that are used, the user can choose them freely. Moreover, The toolset of Torii, Troster and Palma does not supply a storage facility for the results.
6.1.5 Testing Infrastructure

The testing infrastructure, described in [21], supplies the user with all that is needed for experimentation in software testing techniques. Several artifacts, such as software versions, tests, and scripts are included. For instance, to test an automated testing technique, a piece of software with known faults is tested with the testing technique and the score is determined by comparing the found faults with the known faults. The infrastructure provides several tools (e.g., for automation of the tests). An important property of the testing infrastructure is that several versions of one software program are included. This can be used for regression testing.

The similarities between the testing infrastructure and Erix are: (1) the automation of the experimentation process; (2) support for replication; and (3) flexible experiment design support.

The major difference with Erix is the focus on the software testing and regression testing techniques. The domain of the experiments is fixed. Erix can be used to conduct experiments in several domains, including software testing.

6.1.6 Ginger 2

Torii et. al. [49] discuss the Ginger2 environment for Computer Aided Empirical Software Engineering (CAESE). This environment investigates the “human part” of software engineering. All kinds of physical properties of the engineer can be measured and analyzed. This is done by diverse measurement equipment such as an eye tracking system. Ginger2 can be used for tasks as evaluating user interfaces.

Ginger2 is an empirical infrastructure but not in the same way as the previous four. It studies other aspects of software than Erix is designed for. Additionally, the analysis is not done automatically.

6.2 Additional Information

Various other fields in computer science might research techniques that are interesting for automated empirical research, although there is no direct link. Below we mention two: Data mining and coordination languages.

Data mining  Data mining is the extraction of patterns from data using special algorithms [18]. It is used to gain knowledge about the investigated data. Some techniques in data mining are: classification, clustering and summarization.

Data mining is mostly used to analyze large amounts of data. Therefore the analysis is done automatically. The automated analysis of data is a similarity with software measurement research. However, the differences between software measurement research and data mining techniques are numerous. The biggest difference is that data mining may search for a particular pattern in the data whereas software measurement research does not search. Software measurement research collects the data.
Composition languages  Figure 3.1 on Page 26 has similarities with the figures shown at a total other field in software science: compositions languages. Composition languages are used to connect software components. An example of a composition language is Koala [51]. Koala defines the components with the required inputs and the provided outputs. These outputs and inputs need to be connected in order to let the components cooperate. An example from [51] of two components with the inputs and outputs is shown in Figure 6.3 on Page 65.

![Figure 6.3: Two components, C and D, with required and provided interfaces, respectively p and r.](image)

The software tools used in software measurement research can be seen as components of the experiment. Erix and Koala both offer a method to define the components’ input and output. However, Erix is less accurate, no specific output can be defined.

It is clear that Erix and Koala are not on the same level, Erix “components” are complete software programs which can be executed where Koala components are software modules, parts of a program. Although this differences, similar issues in developing and using of the tools might be encountered.
Chapter 7

Conclusions and Future Work

This last chapter reflects on the whole project, what is accomplished and to what degree this contributes to the different fields of the software engineering science.

7.1 Contributions

The introduction mentioned two fields in software engineering which are involved in this project: empirical research and software tools. The contributions are in the same fields.

The contributions of this thesis in the field of empirical research in software engineering are: (1) An overview of how empirical research has been performed and how it is to be performed; (2) Erix, which can help in future empirical research to lower the costs, evolve the field by supporting replication and aggregation; and (3) an overview of automation of the experiments that has been described in the journal *Empirical Software Engineering*.

The field of software tools has been contributed by the given overview of the available research and by the description of the development of Erix. Furthermore, the infrastructure itself is an addition to the field of software tools. Also, Erix can be used in the validation and development of new tools.

In the introduction of this thesis, we have introduced two subjects that can profit from Erix: Software measurement programs and laboratory packages.

7.2 Conclusions

In the past chapters, an overview of empirical research has been given, a special type of empirical research, software measurement research has been introduced and the empirical infrastructure Erix (Empirical Research Infrastructure for Clustered Software) has been described.

Recall the project goal: *find a method to effectively setup a software measurement experiment and conduct this experiment automatically and properly*.

The main result of the project is Erix, the Empirical Research Infrastructure for Clustered Software. To check if Erix meets the project goal, we first need to check if Erix meets its requirements. The requirements are listed and it is specified per requirement if it is met.
7.2 Conclusions

Conclusions and Future Work

- The infrastructure should conduct user-defined experiments automatically. The domain of research that must be defined and conducted with Erix, has been narrowed to the research that can be automated. For this research (software measurement research), three components have been identified: experiment, experiment model and experiment step. These components need to be defined in the research definition. Erix takes care of the execution of the experiment.

- Clustered software measurement research needs to be conducted with the infrastructure. This requirement is met, software measurement research can be setup and conducted (former requirement), and by the introduction of the loop concept a cluster of SUAs can be used as input for the same software measurement research.

- The infrastructure should use standard measurement and analysis software. It must be able to use standard software for measurement and analysis (e.g., gcov). Erix uses shell scripts (builders) to call external software which measure and analyze the needed data. Thus this requirement is met.

- The (interim) results of the research must be stored. The experiment can be defined freely, so the user is free to store all the desired details. Erix manages the location of the output of the steps, consequently, the output can be found easily.

- It must be possible to reuse (parts of) the experiments. The definitions committed to Erix are saved in the definition repository. All definitions in this repository can be reused in new definitions.

- The infrastructure should be OS independent. The infrastructure itself is not OS independent, Erix must be run on a Linux computer. However, Erix can be installed on a network computer, and it can be commanded over the network.

The project goal can be divided into three parts: (1) effective setup of software measurement experiments; (2) conducting software measurement experiments automatically; and (3) conduct proper software measurement research.

Effective setup Erix makes it possible to setup a software measurement experiment by giving an experiment definition. In the definition all the necessary components and actions can be defined according to Erix’ requirements. Additionally, the definitions can be reused in other experiments. Furthermore, the definition language has been evaluated and has been approved. However, some drawbacks of the use of Erix have been identified:

- We have seen that the use of Erix can cost some extra time when the system is deployed. However, this time can be seen as an investment, a research definition repository is being filled by the use of Erix and the reuse of these definition can save time.

- The user of Erix needs to know two languages to define an experiment: The Bash script language and the Nix expression language.

Overall, we conclude that the setup of an experiment with Erix is effective.
Conclusions and Future Work

7.3 Future work

Conduct software measurement experiments automatically  Software measurement research has been defined to specify the empirical research that can be automated. Remark that not all phases of an experiment can be done automatically. The design of the experiment and drawing conclusions cannot be done automatically. The case study shows that Erix can conduct the defined software measurement experiment automatically.

Conduct proper software measurement research  By the use of Erix, two challenges are fulfilled automatically: (1) reduce the costs of empirical research; and (2) support replication and aggregation of the experiments. The other challenges cannot be reached by an automated infrastructure. The researcher should take care of these.

So, with the development and deployment of Erix, the project goal is met. Erix is available at the Swerl website.¹ How to install and use Erix has been described in Chapter 4.

7.3 Future work

Erix can be improved on several points. In the list below, some of these points are given. Probably, by the use of Erix more points can be added to this list.

- The addition of a standard convert module that converts the experiment results to a standard statistics software program (e.g. R²), a opensource variant of the S-Plus package³. R can be used to analyze the measurement results into usable information. The development of such a convert module would probably simplify the setup of new experiments.

- Erix can be equipped with a more sophisticated user interface, which exhibits additional functionality, such as searching to regular expressions and definition fault correction. These kinds of improvements make Erix more usable.

- The debug method, described in Section 5.2, is not user friendly, the user should edit the research definition to get the necessary information. A more user friendly method should be developed.

- Erix can be extended with an export function. Two methods of exporting a research can be distinguished. The first is an export of the definitions, these can be bundled into an definition file. This file can be imported into another Erix installation. The second type of export is the export of the commands of a research. This exported data can be used to conduct the research on a computer without Nix. Both exports are a valuable addition to Erix.

- Simultaneous use of Erix by distinctive users has not been investigated. Concurrency problems may occur, for instance, during the execution of the definition tools. Probably, a locking mechanism needs to be developed to prevent these problems.

¹http://swerl.tudelft.nl/bin/view/Main/ErixInfrastructure
²http://www.r-project.org
³http://www.insightful.com/
Bibliography


### Appendix A

**Glossary**

This appendix gives an overview of terms as used in this thesis. Note that the citations reference to the Appendices bibliography.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analysis tool</strong></td>
<td>A software tool that takes some data to analyze it. The analysis that is done depends on the tool.</td>
</tr>
<tr>
<td><strong>Empirical model</strong></td>
<td>a model of one or more measures and the relations between them. for example more lines of code ⇒ more effort.</td>
</tr>
<tr>
<td><strong>Empirical research</strong></td>
<td>research by observing phenomena.</td>
</tr>
<tr>
<td><strong>Empirical study</strong></td>
<td>a test to compare some theory to one or more observations [63].</td>
</tr>
<tr>
<td><strong>Experiment</strong></td>
<td>An experiment is an investigation. Not to confuse with a controlled experiment (Section 2.2.1).</td>
</tr>
<tr>
<td><strong>Experimental model</strong></td>
<td>an abstraction of the sequence of measurement programs that together make an experiment. This model specifies the implementation of an empirical model.</td>
</tr>
<tr>
<td><strong>Framework</strong></td>
<td>a set of rules and guidelines on one particular subject. A framework does not involve software tools.</td>
</tr>
<tr>
<td><strong>Infrastructure</strong></td>
<td>A toolset that is contains a tool that integrates the other tools.</td>
</tr>
<tr>
<td><strong>Measure</strong></td>
<td>is the number or symbol assigned to the measurement in order to characterize a property.</td>
</tr>
<tr>
<td><strong>Measurement</strong></td>
<td>a mapping from the empirical world to the formal, relational world [24]</td>
</tr>
<tr>
<td><strong>Measurement model</strong></td>
<td>This model reflects a certain viewpoint of the measurement. The model contains some ‘rules’ for the measurement about what to measure and in which unit.</td>
</tr>
<tr>
<td><strong>Measurement tool</strong></td>
<td>A software tool that measures certain properties and returns the results.</td>
</tr>
<tr>
<td><strong>Model</strong></td>
<td>An abstraction of the reality.</td>
</tr>
<tr>
<td><strong>Software engineering</strong></td>
<td>(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1). ([1], page 67)</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Software tool</strong></td>
<td>A computer program used in the development, testing, analysis, or maintenance of a program or its documentation. Examples include comparator, cross-reference generator, decompiler, driver, editor, flowchart, monitor, test case generator, timing analyser. ([1], page 68)</td>
</tr>
<tr>
<td><strong>SUA</strong></td>
<td>Software Unit under Analysis, also called a guinea pig.</td>
</tr>
<tr>
<td><strong>Toolset</strong></td>
<td>a number of collected tools that can be used for a whole group of tasks. The tools can be applied in various combinations to accomplish a certain task.</td>
</tr>
</tbody>
</table>
Appendix B

Empirical Software Engineering Summary

This appendix gives an overview of the collected data of the empirical software engineering journal. Every paper which is published in this journal has been evaluated on several points. These evaluation points are: Quantitative empirical research (yes/no), method (experiment/case study), human interference (yes/no), automatable (yes/no) and automated (yes/no).

Only the real papers has been evaluated, neither the editorial publications nor the special publications such as thesis announces and calls for papers have been evaluated.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Title / Subject</th>
<th>ER</th>
<th>Type</th>
<th>Human actions</th>
<th>Automatable</th>
<th>Automated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Volume 7, 2002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number 1, March</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[11]</td>
<td>Reliability of a single data source</td>
<td>X</td>
<td>Exp</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[62]</td>
<td>Dynamic modeling in UML</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[45]</td>
<td>Evaluation of program slicing for fault localization</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Number 2, June</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7]</td>
<td>Software process improvement motivators</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[13]</td>
<td>Object oriented program comprehension</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[48]</td>
<td>Maintenance effort estimation</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Number 3, September</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[19]</td>
<td>Object oriented technology</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[66]</td>
<td>Understanding of graphs</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[10]</td>
<td>Maintenance effort</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Number 4, December</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[43]</td>
<td>Telecommunication systems</td>
<td>X</td>
<td>CS</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>[77]</td>
<td>Software reliability growth models</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Paper</td>
<td>Title / Subject</td>
<td>ER</td>
<td>Type</td>
<td>Automatable</td>
<td>Automated</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------</td>
<td>----</td>
<td>------</td>
<td>-------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>[46]</td>
<td>Inspection implementation</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Volume 8, 2003</strong> number 1, March</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td>Software process improvement problems</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[29]</td>
<td>Factor Analysis</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[26]</td>
<td>Test management automation</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[51]</td>
<td>Software architecture evaluation</td>
<td>X</td>
<td>CS</td>
<td>-</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Number 2, June</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[65]</td>
<td>Fault threshold prediction</td>
<td>X</td>
<td>Exp</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[76]</td>
<td>The relation between MRE and project size</td>
<td>X</td>
<td>Exp</td>
<td>-</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>[55]</td>
<td>Cost estimation</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[80]</td>
<td>The occurrence of service requests</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Number 3, September</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[40]</td>
<td>Fault prediction Modeling</td>
<td>X</td>
<td>Exp</td>
<td>-</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>[90]</td>
<td>Project success factors</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Number 4, December</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[39]</td>
<td>Software Quality estimation</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[68]</td>
<td>Maintenance distribution</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[64]</td>
<td>Learning effectiveness of simulations</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Volume 9, 2004</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[34]</td>
<td>Testing Techniques</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[18]</td>
<td>Requirement engineering process improvement</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[84]</td>
<td>Usage based reading</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[73]</td>
<td>Knowledge sharing on experiments</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Number 3, September</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[87]</td>
<td>The effect Design pattern on the software’s maintain-</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[15]</td>
<td>Open source change logs</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[89]</td>
<td>Can a survey be used for performance capture</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[41]</td>
<td>Software quality classification</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Number 4, December (special issue)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>Impacts of the organization model on testing</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[33]</td>
<td>Development effort estimation</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[58]</td>
<td>Expert group effort estimation</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[60]</td>
<td>Comparison between Reviews and pair programming</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[71]</td>
<td>An ethnographic study of XP experience</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Empirical Software Engineering Summary

<table>
<thead>
<tr>
<th>Paper</th>
<th>Title / Subject</th>
<th>ER</th>
<th>Type</th>
<th>Human actions</th>
<th>Automatable</th>
<th>Automated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Volume 10, 2005</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number 1, January</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[57]</td>
<td>Expert estimation</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[35]</td>
<td>Documentation within corrective maintenance</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[67]</td>
<td>Comparison between OPM and UML</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[79]</td>
<td>Exploration of the CK metrics suite</td>
<td>X</td>
<td>Exp</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Number 2, April</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[36]</td>
<td>Feedback collection techniques</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[81]</td>
<td>Qualitative method for architecture selection</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[42]</td>
<td>A three group software quality classification technique</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[54]</td>
<td>Impact of team experience on project planning</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[75]</td>
<td>Missing data</td>
<td>X</td>
<td>Exp</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Number 3, July</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[17]</td>
<td>Requirements engineering</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[4]</td>
<td>Transition from case study to class diagram</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[47]</td>
<td>Data collection techniques in software field studies</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[72]</td>
<td>Using metaphors in qualitative data analysis</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[93]</td>
<td>Methodology support in CASE tools</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Number 4, October</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[21]</td>
<td>Empirical infrastructure for testing techniques</td>
<td>X</td>
<td>CS</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>[85]</td>
<td>Characterization of testing techniques</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[86]</td>
<td>Evolvability of information systems</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[83]</td>
<td>Characterizing risky software</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[69]</td>
<td>Research scientists vs. software engineers</td>
<td>X</td>
<td>CS</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Volume 11, 2006</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number 1, January</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[23]</td>
<td>Economics of unit testing</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[20]</td>
<td>Testcase prioritization</td>
<td>X</td>
<td>Exp</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[27]</td>
<td>Workload of web servers</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>[52]</td>
<td>Effectiveness of inspectors</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[82]</td>
<td>The well-being of people using agile development methods</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Number 2, June</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[74]</td>
<td>Evaluation of HOTTest</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[38]</td>
<td>Agile methods in the development of embedded systems</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[12]</td>
<td>Refinement and translation of UML models</td>
<td>X</td>
<td>Exp</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
In the table, an X stands for yes and a - stands for no. Furthermore, Exp stands for experiment and CS stands for case study.

Some notes on the evaluation of the papers:
If the data is taken from a database with data that is collected beforehand, the experiment is not automatable qualitative research is qualified as empirical research, it always needs human actions.

Some doubt has arisen about the semi automatable experiments such as an computer questionnaire. We decided that this kind of research is not automatable because it is not available at any time. If we define a new measure we can take this measure from all available software in contrary to a new question for a questionnaire, the results cannot be collected instantly.

Some papers contain more than one empirical research, for instance, 3 case studies in [2]. We have evaluated it as one empirical research, if one of them is automated, this one is evaluated.
Appendix C

The Nix Expression Language

This appendix briefly describes the Nix Expression, the language used in the experiment definitions. This appendix summaries the and paraphrases parts of chapter 5 of the Nix manual [22]. Moreover, this appendix can be used as a reference.

C.1 A Simple Example

This section introduces a very simple Nix package as an example. This example is used later on to explain several features of the Nix expression language.

The derivation of the package is used to pass the necessary attributes to the builder. A typical derivation is shown in Listing C.1.

```
rec {
    pkgs = (import pkgs/top-level/all-packages.nix) {};

    somevariable = "true";

    inputs = pkgs.stdenv.mkDerivation {
        name = "Erix-1.0";
        src = ./ERIX.tar.gz;
        builder = ./builder.sh;
        inherit somevariable
    };
}
```

Listing C.1: An example of a recursive attribute set containing a derivation

It is also known that the builder is a shell script which uses the attributes passed on by the derivation to execute the needed commands. A builder for the derivation above could look like Listing C.2.

```
source $stdenv/setup

ensureDir $out
tar xvfz $src
```
C.2 Values

In Listing C.1 on Page 83 several attributes are used (e.g., src). These attributes all have a value, the src attribute for instance has the value /asdf/asdfasdf/asd.aas has the type path. The attribute command has the value ‘dir’ that has the type string. Besides these two types, attributes can have also the types Integer, URI and Boolean representing numbers, remote addresses and true or false respectively. All these types are simple values, only one value is represented by the attribute. Nix has two other types of values: lists and attributes sets. A list of values is comparable to a list (or array) in other programming languages. An attribute set contains one or more attributes. Each attribute has its own name and type, it can even be a attribute set itself. An attribute set is indicated with curly braces. An example of an attribute set is shown in Listing C.1 on Page 83 from line 1 to line 12. Special types of attribute sets are emphasized in the following section.

C.3 Language Constructs

To program something useful, only attributes with some value is not enough, some actions must be applied on the values. Nix provides several constructs to do this. In Listing C.1 on Page 83 we have seen some of these constructs, for instance line 1 shows an recursive attribute set, this is an attribute set in which attributes which are “declared” within the attribute set can be used. This is not possible in normal attribute sets.

A function is another language construct (which is very important in a function language). Functions may be defined in Nix, moreover, Nix supplies several built-in functions, these can be called using the builtins. prefix.

Local variables can be defined using a let expression, let <attributes> in <expression>. Attributes can be inherited, this makes the attributes available in another scope. The attribute somevariable is inherited on line 10 in Listing C.1 on Page 83. This makes attribute somevariable available in the builder.
Two constructs that are not shown in the examples of the former section are the conditional and the assertion. The conditional is an if statement to make the outcome of an expression dependent on a certain value, the syntax of a conditional is if e1 then e2 else e3. Assertions are used to check certain value(s), if the values do not comply to the asserted values, the evaluation is aborted. For instance assert e1; e2;, e1 must evaluate to a boolean value. If e1 evaluates to true, e2 will be returned.

Comments are indicated per line with the # character. Multi-line comment is written between /*...*/

C.4 Operators

The Nix expression language provides several operators, these are described in this section.

e.id selects the attribute id from the expression e. e1 e2 calls the function e1 with parameter e2. The operator e ? id tests whether attribute set e contains an attribute id, returns a boolean value.
e1 ++ e2 concats e2 to the list e1.
e1 + e2 concats strings or paths.
To combine attribute sets, the operator e1 // e2 is used.

The boolean operators are !e is a boolean negation. e1 == e2 tests for equality and e1 != e2 tests for inequality. The && and || represent the logical AND and OR. Finally, the logical implication is represented by e1 -> e2.
Appendix D

The Prioritizing Case Study ERIX
Definition

In this appendix the whole definition of the prioritizing case study is presented.

### experiments.nix ###

```plaintext
count=1

bm2006_experiment = (import ./bm_2006.nix) {
    inherit (models) static_vs_dynamic_MTO;
};
#enddefinition
```

### models.nix ###

```plaintext
count=2

static_vs_dynamic = inputs: (import ./static_dynamic.nix) {
    inherit (software) csurf_build csurf_script compile_run gcov
    analysis_static_dynamic pkgs;
    inherit inputs;
};
#enddefinition
```

```plaintext
static_vs_dynamic_MTO = inputs: (import ./static_dynamic_MTO.nix) {
    inherit static_vs_dynamic;
    inherit inputs;
};
#enddefinition
```

### software.nix ###

```plaintext
count=5
```
### experiment ###

```nix
count = 1

filename = bm_2006.nix

{ static_vs_dynamic_MTO }:

rec {
    name = "bm2006_experiment-0.1";
    inputs = static_vs_dynamic_MTO {
        src = /home/af06011/experiment_input/antiword/src;
        inputdir = /home/af06011/experiment_input/antiword/input;
        configfile = /home/af06011/experiment_input/antiword/run-test.conf;
        runcommand = "antiword";
    }
```
buildcommand = "make";
scriptdir = /home/af06011/experiment_input/scripts;
configurecommand = "";
extraCflag = "-DNDEBUG";
}
{
src = /home/af06011/experiment_input/chktex/src;
inputdir = /home/af06011/experiment_input/chktex/input;
configfile = /home/af06011/experiment_input/chktex/run-test.conf;
runcommand = "chktex";
buildcommand = "make";
scriptdir = /home/af06011/experiment_input/scripts;
configurecommand = "configure";
extraCflag = "";
}

#enddefinition

### model ###
count=2
filename=static_dynamic.nix
(
csurf_build, csurf_script, compile_run, gcov,
analysis_static_dynamic, inputs, pkgs):
rec{
name = "Static_vs_Dynamic-0.1";
steps =
let
step1a = interpreter : {
  prog = csurf_build;
  inputs = {
    inherit (inputs) buildcommand src configurecommand maketarget;
    compiler = pkgs.gcc32;
  };
  projectname = inputs.runcommand;
};

step1b = interpreter : {
  prog = csurf_script;
  inputs = {
    inherit (inputs) scriptdir;
    projectname = inputs.runcommand;
  };
};
buildpath = interpreter (step1a interpreter);

step2a = interpreter : {
    prog = gcov;
    inputs = {
        path = /home/af06011/experiment_input/gcov-3.2;
    };
};

step2b = interpreter : {
    prog = compile_run;
    inputs = {
        inherit (inputs) src scriptdir inputdir
        configurecommand maketarget extraCflag
        runcommand configfile;
        inherit (pkgs) perl;
        compiler = pkgs.gcc32;
        gcov = interpreter (step2a interpreter);
    };
};

step3 = interpreter : {
    prog = analysis_static_dynamic;
    inputs = {
        inherit (inputs) scriptdir;
        inherit (pkgs) perl;
        dynamic = interpreter (step2b interpreter);
        static = interpreter (step1b interpreter);
        projectname = inputs.runcommand;
    };
    in
    [step1a step1b step2a step2b step3];
} #enddefinition

filename=static_dynamic_MTO.nix

{ static_vs_dynamic, inputs }:
rec{
    name = "Static_vs_dynamic-MTO-0.1";
    steps =
    let
        step1 = interpreter : rec {
            step1_input = modelinput : {
The Prioritizing Case Study ERIX Definition

    prog = static_vs_dynamic;
    inputs = modelinput;
    loops = map (modelinput : step1_input modelinput)
    inputs;
}

### software ###

    count=10

    filename=csurf/csurf_build.nix

    ( stdenv, inputs ) :

        rec {
            name = "csurf_build-0.1";
            specialbuild = stdenv.mkDerivation {
                inherit name;
                inherit (inputs) src projectname configurecommand compiler
                    maketarget buildcommand;
                builder = ./builder.sh;
            };
        }

    filename=csurf/builder.sh

    source $stdenv/setup
    set -x
    mkdir $out
    PATH=$compiler/bin:$PATH
    echo $configurecommand
    cp -r $src/* ./

    if [ "$configurecommand" != "" ]
        then
            chmod 777 *
            ls -Bl --color=always
            ./configurecommand
    fi
The Prioritizing Case Study ERIX Definition

```
LM_LICENSE_FILE=8752@apsone.st.ewi.tudelft.nl
export LM_LICENSE_FILE

/home/af06011/apps/codesurfer-1.9p6/bin/csurf hook-build
   $projectname -preset-build-options highest --- $buildcommand
   $maketarget
   cp -r * $out

#enddefinition

filename=csurf/csurf_script.nix

{ stdenv, inputs } :

rec {
  name = "csurf_script-0.1";
  specialbuild = stdenv.mkDerivation {
    inherit name;
    inherit (inputs) projectname scriptdir buildpath;
    builder = ./script_builder.sh;
  };
}
#enddefinition

filename=csurf/script_builder.sh

source $stdenv/setup
set -x

ensureDir $out
cp -r $buildpath/* ./

LM_LICENSE_FILE=8752@apsone.st.ewi.tudelft.nl
export LM_LICENSE_FILE
chmod -R ugo+rw *

/home/af06011/apps/codesurfer-1.9p6/bin/csurf -nogui -l $scriptdir
   /csurf/bb.stk $projectname > $projectname.pred

cp -r * $out
#enddefinition

filename=gcov/gcov.nix

{ stdenv, inputs } :

rec {
```
name = "gcov-3.2";
specialbuild = stdenv.mkDerivation {
    inherit name;
    inherit (inputs) path;
    builder = ./builder.sh;
};
#enddefinition
filename=gcov/builder.sh

source $stdenv/setup
set -x
ensureDir $out

cp -r $path/* $out
#enddefinition
filename=compile/compile_run.nix

( stdenv, inputs ) :  
rec {
    name = "compile_run-0.1";
    specialbuild = stdenv.mkDerivation {
        inherit name;
        inherit (inputs) src compiler perl scriptdir extraCflag
        runcommand inputdir configfile configurecommand
        maketarget gcov;
        builder = ./builder.sh;
    };
}
#enddefinition
filename=compile/builder.sh

source $stdenv/setup
set -x
ensureDir prog

PATH=$compiler/bin:$perl/bin:$PATH

if [ "$configurecommand" != "" ]
then
chmod -R ugo+rw *
./configurecommand
fi
313
make CFLAGS="-fprofile-arc",-ftest-coverage",$extraCflag" LDFLAGS=" -fprofile-arc" $maketarget

cd ..
cp $scriptdir/perl/Conf.pm ./
perl $scriptdir/perl/run-tests.pl prog/$runcommand $inputdir
    $configfile results.run $gcov/gcov
ensureDir $out
cp -r * $out
323 #enddefinition
filename=static_dynamic/ranking.nix
{ stdenv, inputs } :
328 rec {
    name = "stat_dyn_rank-0.1";
    specialbuild = stdenv.mkDerivation {
        inherit name;
        inherit (inputs) perl scriptdir static dynamic projectname;
        builder = ./builder.sh;
    };
}
#enddefinition
338 filename=static_dynamic/builder.sh
source $stdenv/setup
set -x
343 PATH=$perl/bin:$PATH
cp $scriptdir/perl/Util.pm ./
perl $scriptdir/perl/stat.pl $static/$projectname.pred $dynamic/
    results.run $projectname.csv $projectname.stats
ensureDir $out
cp -r * $out
353 #enddefinition
Listing D.1: The execution likelihood prioritizing experiment.
Appendices Bibliography


APPENDICES BIBLIOGRAPHY


APPENDICES

BIBLIOGRAPHY


