Automating Acceptance Tests of a Distributed Application

Thesis Project Report

Maarten Abbink
Automating Acceptance Tests of a Distributed Application

Thesis Project Report

by

Maarten Abbink
Automating Acceptance Tests of a Distributed Application

Author: Maarten Abbink
Student id: 4177150
Email: maarten@abbink.me
Defense: April 24th, 2014

Abstract

Designing software tests is oftentimes just as challenging as designing the system under test (SUT) itself. A complex SUT will require a complex test suite to achieve a substantial amount of test coverage, and to instill confidence in its correctness. In order to engineer complex systems, software engineers frequently apply patterns that are implemented in reusable libraries. In testing this is no different; by applying reusable patterns, engineers are able to abstract over complex systems, and create concise and expressive test programs.

This work presents a technical solution to the acceptance testing challenges of a business case. By providing a framework for constructing distributed test programs, we aid engineers in creating concise acceptance test programs for a particular SUT. We thereby contribute to the test automation effort of the business case at hand.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty of EEMCS, TU Delft
University supervisor: Dr. ir. F.F.J. Hermans, Faculty of EEMCS, TU Delft
Committee Member: Dr. C. Hauff, Faculty of EEMCS, TU Delft
Preface

Fugro Intersite B.V., a subsidiary of Fugro N.V., develops hardware and software for hydrographic (off-shore) surveying. One of their products is Starfix.NG, a complex software system used for planning and executing hydrographic surveys. Starfix.NG aims to support surveyors aboard a vessel in navigating, configuring equipment and collecting data. The software runs distributed across multiple workstation computers and serves different users’ needs simultaneously. The acceptance testing effort in the development of Starfix.NG largely consists of manual testing. This work presents a technical solution named DistrEx, that aids the transformation of manual testing protocols into automated testing programs. An in-depth discussion of the business case at hand is provided in Ch. 2.

I would like to express my gratitude towards my coworkers at Fugro, for their help and involvement in building DistrEx, and their sincere interest in my work. I also would like to thank my friends for their support during my MSc thesis project.

Maarten Abbink
Delft, the Netherlands
April 10, 2014
# Contents

**Preface**  iii  

**Contents**  v  

1 Introduction  1  

1.1 Business Case  2  

1.2 Software Testing  2  

1.3 Distributed Systems  2  

1.4 Outline  3  

2 Starfix.NG  5  

2.1 Technical Characteristics  5  

2.2 Software Engineering Characteristics  7  

2.3 Challenges  16  

3 Software Testing  19  

3.1 Test Automation  20  

3.2 Acceptance Testing  20  

4 Distributed Systems  23  

4.1 Sequential Systems  23  

4.2 Concurrent Systems  23  

4.3 Testing Distributed Systems  25  

4.4 Engineering Distributed Acceptance Tests  25  

5 DistrEx  27  

5.1 Notation Requirements  27  

5.2 Semantic Primitives  28  

5.3 Implementation  36  

5.4 Evaluation  52
CONTENTS

6 Conclusions and Future Work 53
   6.1 Conclusions .......................................................... 53
   6.2 Future Work ............................................................. 55
   6.3 Threats to Validity ..................................................... 56

7 Related Work 59
   7.1 Automated Testing .................................................... 59
   7.2 Concurrent Computing ............................................... 60

Bibliography 63

A NG Codebase Metrics 69
   A.1 Complexity ............................................................. 69
   A.2 Test Coverage .......................................................... 69
   A.3 Contributors' Involvement .......................................... 70
   A.4 Miscellaneous Remarks ........................................... 71

B Glossary 73

C Acronyms 81
Chapter 1

Introduction

At a software engineering conference, participants were given an awkward question to answer: “If you had just boarded an airliner and discovered that your team of programmers had been responsible for the flight control software, how many of you would disembark immediately?”

Among the forest of raised hands only one man sat motionless. When asked what he would do, he replied that he would be quite content to stay aboard. With his team’s software, he explained, the plane was unlikely to even taxi as far as the runway, let alone take off.

The true nature of this anecdote may be disputed, but it is true in one respect: Developing nontrivial software systems that are free of faults is hard. With growing complexity comes the risk that some part of the software system is not functioning correctly. In order to minimize this risk, software engineers most commonly resort to testing [73, 42]. The goal of software testing is to reveal any faults that a software system might contain, so they may be documented, or better yet, corrected in time before delivering the system. However, software testing itself is difficult as well, which has led to the creation of a wealth of resources addressing a wide range of challenges presented by the software testing discipline [13].

As the title of this report (Automating Acceptance Tests of a Distributed Application) suggests, in this work we are primarily interested in automated acceptance testing, applied to distributed systems. Therefore, the following chapters will predominantly draw from previous work in the fields of automated testing and distributed systems. The focus on this combined area of interest is motivated by a business case that offers numerous open software engineering challenges, in particular those related to automated acceptance testing of a distributed application (see Sec. 1.1). The following chapters report the analysis of a distributed software system with respect to automated acceptance testing, solutions to a subset of the subsequently discovered challenges it faces, the implementation of those solutions, and finally the application of the solutions to the system.
1. Introduction

1.1 Business Case

The context of this work is the offshore surveying software system Starfix.NG (NG) created by Fugro Intersite B.V. (Intersite). The main component of the NG system is a graphical application (the NG application) of which multiple instances can operate collaboratively on a group of networked computers. User interactions with the application generally have a non-local effect, i.e. interactions may cause remote application instances – which are connected through the computer network – to react. This design characteristic of the NG application introduces additional challenges in acceptance testing that cannot be addressed by employing conventional testing techniques alone. In part, these challenges are nontrivial to overcome, given the complex nature of the NG system as a whole. Besides the main NG application, numerous precisely configured system settings and auxiliary applications are required for the NG system to operate properly. Chapter 2 discusses technical details of the business case, focusing on the NG application.

1.2 Software Testing

An important part of software development is making sure that the software under development functions correctly, i.e. that the software is free of faults. This matter of quality assurance is most commonly addressed through testing, which is a way of establishing correctness through experimentation [13, 49]. The formal contender to testing is proving that an implementation satisfies certain mathematical properties.

Since testing plays an important role in software engineering, this work will concentrate on testing as well. The importance of testing is best demonstrated by the fact that oftentimes over 35% of resources in software development are directed towards testing [33]. Therefore, industry and academia are continuously exploring ways of cutting testing costs, all the while improving the results of testing efforts. In particular, forms of testing beyond the unit testing level offer a wealth of opportunities for improvement, as these forms are less well understood or are more labor-intensive to realize. Examples of such higher level forms of testing are integration testing and acceptance testing, where software behavior is verified on an increasingly conceptual level.

In order to achieve maximum utility of the ideas presented in this work, we strive to meet the needs of the business case at hand. It is for this reason that we will address open challenges in the area of acceptance testing with one particular distributed system in mind; that of the business case.

1.3 Distributed Systems

Distributed software systems are systems that run on multiple machines simultaneously. The machines that make up such a distributed system are connected through a network, over which information can be exchanged. The main difficulty of distributed systems is that events occurring on different machines happen simultaneously. By itself, this concurrent behavior cannot be ordered, meaning that no total order can be assigned to the global set of
concurrent events [39, 56]. The same holds true for concurrent systems in general, of which distributed systems are merely a specific subset.

Simultaneously occurring events are neither impossible to reason about, nor impossible to implement. However, contemporary programming languages provide limited facilities for specifying concurrent behavior. Moreover, the facilities that do exist for this purpose tend to obfuscate the code, masking the programmer’s intentions, or worse, open the door for faults that are hard to reveal and correct [41].

When discussing distributed systems, we keep the business case at hand in the back of our minds, in order to decide how to address arising challenges in the domain of distributed systems. Hence, solutions presented here strive to be primarily applicable in the context of the business case.

1.4 Outline

In the previous sections the reader has briefly been introduced to testing. Next, Ch. 2 will introduce the business case around which this work revolves. Ranging from technical details to open challenges of the business case, the subject of Ch. 2 is ultimately condensed into a concluding articulation of research questions. These research questions are explored and answered in the remainder of this work.

Chapter 3 and Ch. 4 discuss the fields of software testing and concurrent systems, respectively. Leaning on the background knowledge provided by previous chapters, Ch. 5 embarks upon formalizing observations made in preceding chapters. Through a number of proposed solutions under the collective name DistrEx the foundation for addressing a selection of the business case’s open challenges is laid down.

Chapter 6 summarizes the results of this work, as well as subjects that remain open for future work. Finally, related work is presented in Ch. 7.
This chapter discusses the technical details of the Starfix.NG software system. An overview of Starfix.NG’s technical characteristics is provided in Sec. 2.1, followed by an overview of software engineering characteristics, such as contributors’ involvement, in Sec. 2.2. Section 2.3 concludes this chapter by articulating the challenges the Starfix.NG software system faces as research questions, which will be addressed throughout subsequent chapters.

Starfix.NG is a software system for offshore surveying. Developed by Intersite, it is exclusively employed aboard the M.V. Fugro Galaxy (Galaxy) vessel to plan and execute offshore surveys. An offshore survey involves sailing along a number of precisely defined paths, during which sensor data from surveying equipment is collected and visualized on screen. Since multiple computers can execute Starfix.NG simultaneously, the system qualifies as a distributed system. In such a configuration, each workstation has its own specific set of data acquisition and presentation tasks which is required for both an overall successful survey execution, and supporting different users in their operational duties.

2.1 Technical Characteristics

2.1.1 Data Acquisition

Surveying equipment produces sensor data that needs to be collected and integrated to become useful. Typically, offshore surveys involve the use of a Global Navigation Satellite System (GNSS) receiver, gyroscope, accelerometer, altimeter and magnetometer, in addition to hydrographic equipment. Starfix.NG supports a variety of data input methods: RS-232, Universal Serial Bus (USB) and Transport Layer network communication. In case of the latter, surveying equipment commonly sends its data to a predefined destination on the network, using streams of User Datagram Protocol (UDP) data. Equipment for relaying RS-232 data to an ethernet network is often employed as well, which allows for more flexibility in configuring a survey’s hardware setup. Figure 2.1 shows an example setup involving numerous sensory equipment devices, e.g. radio signal receivers, a high precision clock, and numerous (hydrographic) measuring devices.

Data generated at the producer (e.g. sensor) is delivered to the Starfix.NG application in a timely manner, where it is decoded as quickly as possible. Once decoded, units of data are
2. Starfix.NG

Figure 2.1: Starfix.NG example hardware setup. An abstract representation of how sensory equipment is connected to the Starfix.NG application.

Published for any connected Starfix.NG application instance to process. The processing of decoded data happens as quickly as possible, but since the steps along the path from data delivery to processing are not implemented as Real-Time systems, Starfix.NG does not strictly qualify as a Real-Time system.

Once a unit of data is processed, the result is recorded in a persistent storage system. From here, other application instances can once again access the processed data, regardless of the application instance the processed data originates from.

2.1.2 Components and Environment

The main component of the Starfix.NG system is the Starfix.NG application. The remaining components of the system are auxiliary applications surrounding the main Starfix.NG application, which — along
Software Engineering Characteristics

with the operating system’s configuration — constitute the application’s environment.

The NG application, written in the C# 4.0 programming language, is targeted at the Microsoft Windows 7 operating system for x86-64 processors. Additionally, the .NET Framework (.NET) versions 3.5 and 4.0 and the VS2005 and VS2010 runtime libraries are required dependencies. Auxiliary applications surrounding the NG application are required for proper operation of the system. These are the proprietary message-passing middleware MessageManager (MM) and a Network Time Protocol (NTP) client program. A distributed setup of multiple NG application instances additionally requires that the computers involved be connected through a Local Area Network (LAN) and have common access to a shared file system that supports the locking of files.

MM is not only used for inter-instance communication, but also for intra-instance communication, i.e. different subsystems of one NG application instance are also connected through MM. The configuration and processed data for a specific surveying task are written to the (shared) file system, primarily in SQLite files. In case of a distributed NG setup, this part of the file system must be accessible to all NG application instances, which calls for the aforementioned shared file system. In practice, the shared file system solution in use is always Microsoft’s Server Message Block (SMB) implementation.

2.2 Software Engineering Characteristics

In this section we will consider three aspects of the NG application from a software engineering perspective: Codebase metrics, the composition of software components and how software testing is employed within the codebase.

2.2.1 Codebase Metrics

The codebase of the NG application consists of approximately 2,304 Kilo Lines of Code (KLOC), roughly 1,802 KLOC (78%) of which are located in C# code files. Of the C# code files, over 706 KLOC (39%) are identified and considered coverable by OpenCover during test runs. OpenCover ignores empty lines, comments, and other non-behavioral code. Altogether, OpenCover test runs identify 76,196 methods, spanning 10,466 classes, and 110 assemblies. while in total, there are 26 more assemblies that are part of the NG application. No further details are available for these 26 assemblies, as they are not analyzed by OpenCover. The aforementioned 110 assemblies collectively depend on 161 additional assemblies, 67 of which are developed by Intersite. The remaining 94 dependency assemblies are externally acquired Open Source Software (OSS) and Commercial Off-The-Shelf (COTS) products. Table 2.1 summarizes the facts presented in this paragraph.

The following subsections discuss parts of this information in more detail. Details about how the information was retrieved are provided in App. A.

---

1This information is presented separately to benefit this chapter’s textual structure.
2. **STARFIX.NG**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes per assembly</td>
<td>2</td>
<td>968</td>
<td>95.0</td>
<td>37</td>
</tr>
<tr>
<td>Methods per class</td>
<td>1</td>
<td>880</td>
<td>7.2</td>
<td>3</td>
</tr>
<tr>
<td>Cyclomatic complexity of methods</td>
<td>1</td>
<td>67</td>
<td>1.8</td>
<td>1</td>
</tr>
<tr>
<td>Coverable LOC per assembly</td>
<td>12</td>
<td>53,491</td>
<td>6,424.0</td>
<td>2,662</td>
</tr>
<tr>
<td>Coverable LOC per class</td>
<td>1</td>
<td>3,641</td>
<td>67.5</td>
<td>24</td>
</tr>
<tr>
<td>Coverable LOC per method</td>
<td>1</td>
<td>460</td>
<td>9.2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2.1: An overview of the codebase metrics.

### Complexity

Considering, again, results collected during OpenCover test runs, we observe the code complexity to be as follows. Note that by complexity, here we are referring to a code unit’s size, indicated by the number of child units it contains. The size of different units of code varies widely per assembly. The number of classes per assembly ranges from 2 classes to 968 classes, with a median of 37 classes per assembly. This includes anonymous inner classes. Similarly, the number of methods per class is divergent as well, ranging from 1 method to 880 methods per class, with a median of 3 methods. OpenCover further reports cyclomatic complexities of methods between 1 and 67, with a median of 1. This includes anonymous methods, as well as getter and setter methods. Classes and methods comprise 1 to 3,641 and 1 to 460 lines of coverable code, respectively, with medians of 24 and 4. Table 2.2 lists the summarizes the facts presented in this paragraph.

### Test Coverage

In the development process, currently two test coverage metrics are considered: Unit test coverage, and automated acceptance test coverage. Within the domain of each test run, we observe a very polarized distribution of line coverage percentages. During unit testing, of all 76,196 methods, 31,619 (41.5%) achieve 100% line coverage, 3,924 (5.1%) more achieve at least some degree of coverage, and the remaining 40,652 (53.3%) methods are not covered. Analogously, during acceptance tests, 26,746 (35.1%) methods achieve
(a) Methods’ line coverage by test run.

<table>
<thead>
<tr>
<th>Coverage Type</th>
<th>Unit Testing</th>
<th>Acceptance Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% coverage</td>
<td>31,619</td>
<td>26,746</td>
</tr>
<tr>
<td>Some coverage, but not 100%</td>
<td>3,924</td>
<td>3,012</td>
</tr>
<tr>
<td>0% coverage</td>
<td>40,652</td>
<td>46,438</td>
</tr>
</tbody>
</table>

(b) Methods’ combined line coverage.

<table>
<thead>
<tr>
<th>Coverage Type</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covered by unit tests only</td>
<td>6,085</td>
<td>7.9%</td>
</tr>
<tr>
<td>Covered by acceptance tests only</td>
<td>299</td>
<td>0.3%</td>
</tr>
<tr>
<td>0% coverage</td>
<td>40,353</td>
<td>52.9%</td>
</tr>
<tr>
<td>Jointly covered</td>
<td>3,924</td>
<td>5.1%</td>
</tr>
<tr>
<td>More extensively covered by unit tests</td>
<td>855</td>
<td>1.1%</td>
</tr>
<tr>
<td>More extensively covered by acceptance tests</td>
<td>1,969</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

Table 2.3: An overview of the [NG] application’s test coverage. Percentages denote the number of methods relative to the total number of methods.

100% line coverage, 3,012 (3.9%) more are covered to at least some degree, and the remaining 46,438 (60.9%) methods are not covered. Globally, 40,353 (52.9%) methods remain untouched by either test run. In total, during acceptance testing, 299 (0.3%) methods covered that are not covered during unit testing, and during unit testing, 6,085 (7.9%) methods are covered that are not covered during acceptance testing. The union of methods covered during both test runs contains 29,459 (39.6%) methods. Within this set, 1,969 (2.5%) methods are covered more extensively during acceptance testing, and 855 (1.1%) methods are covered more extensively during unit testing. Table 2.3 summarizes the facts presented in this paragraph.

A more detailed view on the collected data is presented in Fig. 2.2. For each assembly in the [NG] application, the figure displays the number of coverable lines within that assembly, and for each line whether it was covered by unit tests and/or acceptance tests. From this it becomes apparent that, compared to unit tests, acceptance tests add to test coverage significantly. Note that due to confidentiality concerns, assembly names have been replaced with aliases.

From the presented data, we observe that test coverage during both unit testing and acceptance testing is very polarized: Either a method is thoroughly tested, or it is hardly tested at all. Reportedly, management is responsible for this polarization by demanding that all methods added after a certain point are to be maximally covered by unit tests. Consequently, less attention is paid to address lacking unit test coverage for pre-existing methods. It is noteworthy that the presented data also comprises information on methods for which testing does not necessarily make sense, e.g. getter and setter methods. These will add some degree of distortion to the presented figures.

The current manual acceptance testing effort applied in the [NG] development process does not yield any code coverage information.
Figure 2.2: Test coverage in comparison. The vertical axis lists all assemblies. The widths of colored sections indicate the number of lines in the corresponding assembly that were covered exclusively by unit tests, exclusively by acceptance tests, by both kinds of tests, or not at all.
Contributors’ Involvement

The previous subsections describe the size of the NG codebase, as well as the key characteristics of the automated testing effort, as reflected by the code. This subsection will present contributors’ individual contributions to the codebase.

In order to produce the data for this section, we have collected information from the NG application’s Git source code repository. We have considered all contributions in the time frame between March 4th, 2012 and March 4th, 2013. During this time the repository has recorded contributions from 51 authors to 25,375 different source code files. Of those files, 6,227 contain test code. Overall, individual files were authored by between 1 and 16 authors, with a median of 2 authors per file. The same metrics apply to the set of non-test code files. Test code files alone are edited by between 2 and 11 authors, with a median of 2 authors. By grouping files by the assembly they belong to, we find that all 139 assemblies are modified by between 1 and 36 authors, with a median of 8. When considering only test code files on a per-assembly basis, the data reveals that 108 assemblies contain test code files. These assemblies are contributed to by between 1 and 33 authors each, with a median of 6.

Individual contributors have authored between 1 and 16,328 files each, with a median of 386 files per author. By again separating test source code files, we observe that authors have contributed to 1 to 3,763 test files, with a median of 287. Considering files’ assignment to assemblies, contributors have worked on 1 to 136 assemblies, with a median of 19 assemblies per author. Finally, when considering only test files on a per-assembly basis, we find that 46 authors worked on 1 to 102 assemblies, with a median of 13 assemblies per author.

A visual representation of the collected data is provided in Fig. 2.3. The matrix in Fig. 2.3 reveals how involved each contributor was in evolving each assembly’s test code and application code. The colors in the depicted matrix represent values according to the schema presented in Fig. 2.4. Note that due to confidentiality concerns, assembly names and contributor names have been replaced by aliases. Additionally, assembly name aliases in Fig. 2.3 are unrelated to assembly name aliases of Fig. 2.2. The facts in this subsection are summarized in Tbl. 2.4

The data presented above demonstrates that contributors have limited contributing experience throughout the codebase. We claim that development activities building on the work contained in certain source files will eventually lead to changes in those files. The rationale for this claim is that achieving the best possible result of a development activity requires existing code to be altered. This can mean e.g. that existing code might require refactoring in order to be reusable for a new feature, or that documenting comments are altered. Additionally, we assume that contributing to a source code file happens only when contributors are confident about their own understanding of the affected code. Taking the previous claim and this assumption into account, our data shows that all 51 contributors have limited knowledge of the codebase as a whole. Even when considering the data on a per-assembly basis, the same case can be made. Moreover, we assert that changes to test code do not reflect proportionate involvement of all contributors.
2. **STARFIX.NG**

(a) **Contributors per unit.**

<table>
<thead>
<tr>
<th>Unit</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>All files</td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Test code files</td>
<td>2</td>
<td>11</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Assemblies</td>
<td>1</td>
<td>36</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Assemblies’ test code files</td>
<td>1</td>
<td>33</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

(b) **Units individual contributors are involved in.**

<table>
<thead>
<tr>
<th>Unit</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>All files</td>
<td>1</td>
<td>16,328</td>
<td>966</td>
<td>386</td>
</tr>
<tr>
<td>Test code files</td>
<td>1</td>
<td>3,763</td>
<td>287</td>
<td>129</td>
</tr>
<tr>
<td>Assemblies</td>
<td>1</td>
<td>136</td>
<td>27</td>
<td>19</td>
</tr>
<tr>
<td>Assemblies’ test code files</td>
<td>1</td>
<td>102</td>
<td>17</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 2.4: An overview of contributors’ involvement in the **NG** application’s codebase.

2.2.2 **Software Component Composition**

The **NG** application is composed of layered building blocks from three primary origins. Figure 2.5 illustrates this layered composition of building blocks.

Each building block represents a group of assemblies that provides conceptually related functionality. The building blocks’ origins are either **NG** internal, **Intersite** internal, or external. **NG** internal building blocks contain assemblies that are developed as part of the **NG** application’s source code repository. **Intersite** internal building blocks contain assemblies that are developed by **Intersite** for the sole purpose of capturing business logic, specific to the offshore surveying application domain, in a reusable way. In practice, these building blocks are used solely in the **NG** application, but their development process is strictly separate. The third group of building blocks, named external, comprises all 3rd party components. These include standard target platform system libraries, as well as externally acquired **COTS** and **OSS** libraries.

An informal survey among **NG** contributors revealed that currently no consensus about a precise conceptual decomposition of the **NG** application exists. The depiction thereof provided in Fig. 2.5 is the result of informal elicitation among contributors. While tools capable of extracting architectural diagrams from code are available, a reported lack of an architecturally sound system design is likely to inhibit the generation of a meaningful decomposition.

2.2.3 **Testing**

A summary of **NG**’s testing effort has been provided in Sec. 2.2.1. This summary revealed that currently three types of testing are employed in the software development process of **NG**: Unit testing, manual acceptance testing, and automated acceptance testing. From this
Figure 2.3: Contributors’ involvement per assembly. The vertical axis lists all assemblies. The horizontal axis lists all contributors; once for their contributions to application code and once for their contributions to test code. Colored cells inside the matrix indicate for each combination of assembly and contributor what their accumulative contribution is in the respective area of application code and test code. Figure 2.4 explains the semantics of the colors used.
2. **Starfix.NG**

![Color index for contribution sizes](image-url)

Figure 2.4: Color index for contribution sizes. Different colors are used for application code and test code. Colors are used in Fig. 2.3.

![NG architecture](image-url)

Figure 2.5: NG architecture. An approximate representation of how the NG application is composed. Each block represents a group of assemblies/libraries. A block’s color denotes its origin: blue denotes an NG internal block, green denotes an Intersite internal block, and purple denotes an external block. The distinction follows that integration testing is not employed at a significant scale, within the realm of NG. The following subsections will discuss different kinds of automated testing in greater detail.

**Unit Testing**

On average, unit tests do not achieve a very high grade coverage, i.e. 53.3% of all methods are not covered by unit tests. However, classes, namespaces and in part also assemblies, that are tested using unit tests, are often extensively tested, resulting in a significantly larger test coverage that often exceeds the 85th percentile. This leads to a distribution of coverage data over two extremes: Highly covered and poorly covered sections of code.

The cause of this (negative) concentration of unit test coverage reportedly lies in management decisions. Management dictates a certain minimum degree of unit test coverage for some sections of the code, while other sections remain free from any such impositions.

---

2Personal correspondence.
Additionally, we are unaware of any test adequacy criterion \cite{73} being used for coordinating test designers’ efforts.

**Integration Testing**

As mentioned above, integration testing is hardly practiced at all within the development process of NG. Integration testing reportedly occurs as part of the unit test suite, although, only sparsely.

One reported caveat of the few existing integration tests is inadequate faking and mocking. Faking and mocking are techniques used to simulate a part of a system such that it interacts realistically with depending components under test, while limiting the number of interactions with other components. In NG, for some components there are multiple independently developed and maintained test implementations. This diversity in implementations effectively reflects a diversity in interpretations of a component’s specification. As a result, tests involving different fake or mock implementations of the same component cannot be guaranteed to have equivalent test contexts.

Moreover, contributors’ accounts state that most components are currently ill-suited for integration testing, meaning that sensible integration test cases cannot be formulated. This is caused by insufficient understanding of how components interact in practice, and — by design — the inability of a test program to examine a component’s internals from the outside. The latter inhibits both initializing components to a desired state and verifying their state for the sake of testing.

**Acceptance Testing**

Automated acceptance testing of NG follows manual testing protocols. These protocols consist of a sequence of steps that have to be executed by a tester. Each step is either an instruction telling the tester to interact with the System Under Test (SUT) in some way or another, or an instruction for the tester to verify that the SUT is in a desired state.

These protocols are transformed into automated test cases by translating each manual instruction into a partial test program that manipulates and inspects the SUT’s Graphical User Interface (GUI). By leveraging Microsoft’s Windows Automation API \cite{47}, these partial programs simulate manual interactions with, and observations of, the SUT’s GUI. Such translated instructions are combined to form complete test programs that resemble automated instances of test protocols.

This arguably high-level approach to automating acceptance tests introduces additional difficulties that software developers need to take into account when working on the SUT. In particular, the GUI requires special preparation in order for a test program to simulate manual interactions. Additionally, asynchronous internal behavior of the SUT is difficult to observe from the GUI alone. Such added difficulties increase the risk of an acceptance test program being faulty, because of the additional test code required to overcome them.

Considering the technical challenges presented above, we reason that despite the additional difficulties involved in automating acceptance tests, this form of testing is currently the most accessible form of automated testing in NG beyond the unit level. The main ar-
argument supporting this statement is that acceptance testing — in status quo — requires no changes to the SUT as opposed to, e.g. integration testing.

2.3 Challenges

The main symptoms that the development of NG suffers from, are the rather large size of its codebase in combination with inadequate automated testing. Consequently, any modification made to the code might introduce new faults that will not be discovered by the existing automated test suite. Additionally, it is shown that the codebase is too extensive for most contributors to fully comprehend. We argue that with a limited familiarity and understanding of the codebase, contributors cannot properly foresee the extent to which change affects the proper operation of dependent code across the codebase. The latter of which makes the lack of adequate automated tests even more apparent.

These symptoms, however, cannot simply be addressed head on by extending the suite of automated tests. This goal is difficult to pursue, since it requires a fundamental understanding of the SUT’s design from which a test model can be derived [26]. Yet, there is currently no consensus among contributors about the NG application’s internal design, nor are there design documents available that specify components’ requirements in a testable way.

Even if such consensus were to be achieved, and the system’s internal design were sufficiently documented and understood, changes to the code in order to facilitate better testing would be required first. At this point, we have come full circle: Making changes requires better testing, but better testing requires making changes. Since the NG system has been released and is in use, continuous maintenance is required, making the demand for better testing less pressing then the demand for (bug fixing) changes. After all, serious malfunctions that jeopardize productivity must be repaired first.

Currently the only approach to safeguarding and increasing the codebase’s quality beyond unit testing, is through acceptance testing. Based on the premise that existing use case documents are testable through acceptance tests, it is at this level that NG is currently most testable. Additionally, a more precise system specification in the form of and extensive test suite will aid efforts of correctly modifying the existing SUT. Subsequently, increased confidence in modifications’ quality would promote changes that yield better testable components, which in turn are a prerequisite for extending the integration test suite.

Lastly, in order to best leverage the available testability, we expect to achieve the best possible results in improving and safeguarding the system’s quality through automating the acceptance testing process to the fullest extent possible.

2.3.1 Research Questions

The challenges explained above will be rephrased as research questions. By then seeking answers to these questions, we will address the underlying challenges.

Research Question 1 Does acceptance testing retain its utility if integration tests are lacking?
Using anecdotal evidence from the business case that is NG and an examination of test coverage reports of the NG application, we will develop an answer to this question. Assuming that R.Q. 1 will be answered positively, we continue:

**Research Question 2**  *Can meaningful acceptance tests for distributed systems be formulated?*

This question comprises a multitude of subjects, for which we will articulate separate research questions below. If we can find answers to each of these follow-up research questions we argue that we have answered R.Q. 2.

**Research Question 2.1**  *How can a test program interact with a distributed system?*

By first explicating how test programs interact with local systems, we seek to identify the steps required to apply such interactions to remote systems.

**Research Question 2.2**  *Do such interactions leave test validity intact?*

Interactions with the SUT might alter its behavior. It is therefore important to establish that any such alterations do not compromise test validity, i.e. despite potentially different behavior of the SUT during test, tests still establish the same validity properties.

Lastly, we seek to articulate answers to R.Q. 2.1 and R.Q. 2.2 in a reusable form:

**Research Question 2.3**  *Can a notation for writing distributed acceptance tests be developed?*

We seek to answer this question by proposing a notation of which proper usage guarantees the creation of test programs capable of interacting with distributed systems. Furthermore, this notation will keep its users from creating concurrency bugs.
Chapter 3

Software Testing

This chapter discusses software testing as sub-discipline of software engineering. Particular attention is paid to test automation (Sec. 3.1) and acceptance testing (Sec. 3.2).

Software testing is the discipline of verifying behavioral correctness of a software system, the SUT, through experimentation [13]. For brevity, within software engineering the term “software testing” is often substituted by “testing”.

Testing is an integral part of software engineering, which is not only relevant until after the SUT has been fully developed, but throughout the entire software engineering process testing has a strong influence on development. The strongest advocates of so-called Test Driven Development (TDD) even go as far as to start by exclusively developing tests. Only once a software unit has been developed within test code will it be refactored and moved to become part of the actual SUT [8]. TDD in its pure form occurs rarely in practice. Instead, practitioners often find a balance between pure TDD and testing software units after they have been developed.

Testing involves the execution of a number of test cases, which together constitute a test suite. A test case, often abbreviated to plainly “test”, is an experiment that aims to verify the behavioral correctness of the SUT in — ideally — one aspect only. The distinction between correct and incorrect behavior is determined by a functional specification that the SUT must follow. The aspects addressed by a test case are said to be covered by that test. Together, the test cases of a test suite have a combined coverage, i.e. an accumulated set of aspects that the suite collectively covers. From the ideal that tests cover at most a single aspect follows that a test should only be concerned with a subset of the SUT’s behavior. By extension, this means that executing a test will only involve the execution of a subset of the SUT’s code. The heart of any testing effort is therefore to identify an ideal subset of aspects to test, such that their corresponding tests collectively cover enough aspects with minimal effort.

We casually define a test case as a triple \( \langle B, I, O \rangle \). \( B \) specifies the behavior to be tested, i.e. the subsystem of the SUT that the test experiment will work on. \( I \) represents the input stimuli that will be fed to \( B \) through the course of the test experiment. \( O \) denotes the outputs that \( B \) is expected to produce. The actual outputs that are observed during test execution are compared to \( O \) in order to assert that \( B \) behaves either correctly or incorrectly.

The test execution process involves executing all test cases in a suite and collecting the verdict that each test case finds. The resulting collection reveals which parts of the system...
pass the tests they are subjected to and can therefore be considered correct, and which do not and must therefore be considered incorrect. No such conclusive claims can be made about (sub)systems that are not tested.

3.1 Test Automation

Following the argument of Gelperin and Wilburn [31] we argue that testing in general yields optimal results if the process is as extensively automated as possible. The step in the testing process that is arguably most suitable for automation is test case execution, and it is this concept that is meant by ‘test automation’.

The rationale is straightforward: An automated process, as opposed to a manual process, will be error free, executes faster, and requires no human intervention to yield the desired results. In order to realize automated testing, test programs must be capable of both predictably interacting with the SUT and predictably deciding whether the SUT behaves correctly. Moreover, being able to uncomplicatedly decide upon the SUT’s correctness is paramount in the face of automation, as a test program does not have the intelligence of a manual, human tester to arrive at such a conclusion. To ease satisfying this requirement and thereby ease the creation of test programs, the SUT must be designed and implemented with testability in mind [12].

3.1.1 Test Execution

In practice, test execution is determined by how tests are formalized. Most commonly, automated tests are formalized as methods of specially purposed test classes. These methods interact with parts of the SUT as if they were regular program code, e.g. by calling the SUT’s methods through the regular method invocation mechanism and processing their return values. The SUT’s return values — or raised errors — are then compared to expected outputs, from which a verdict regarding correctness is derived. Test methods are executed by a generic test execution program. Such a program can access test code without any built-in knowledge about any specific test method to execute. Note that this commonly practiced approach to testing does not involve interacting with the SUT as a separately running process. Instead, parts of the SUT are merely executed within the lifecycle of the test program’s execution, on an as-required basis determined by the trace of nested method invocations originating from the test method.

3.2 Acceptance Testing

In software engineering, acceptance testing is the process of determining whether or not the entire SUT meets its requirements, as opposed to just testing a small subsystem [44]. In practice, it involves using all the features implemented by the SUT and checking whether they conform to their specification. In case of a reactive system that is operated through a User Interface (UI), this means using available UI interactions to accomplish the task of using each feature.
Acceptance Testing

The specifications on which acceptance test are based are commonly user stories \cite{24}, or use cases \cite{38} that dictate which tasks the user needs to be able to accomplish using the SUT and how \cite{17}. When automating acceptance tests, each test case is translated into a test program that mimics these interactions. From a technical viewpoint, a program's UI is just that: An interface for users. A UI is only required, since users cannot invoke system functionality without it. In order to better support the user, UIs – particularly GUIs – are often more than just a collection of input and output mechanisms wrapping the underlying system: UIs often contain interface behavior, such as input validation, as well.

Since unlike users, programs can invoke system functionality without a UI and UIs contain additional behavior, acceptance test programs should avoid having to go through the UI. Instead, an alternative interface, purpose-built for testing, should be used by acceptance test programs. UI behavior and the connection between the UI and the underlying system can be tested separately. This concept is known as a test bus imperative \cite{43}. Despite potential advantages that the test bus imperative might bring in areas such as the separation of concerns within test design, it requires that the SUT's design supports it. However, not all systems support this, making the approach that involves testing through the UI the only available option. While cumbersome, this approach can be streamlined still, using UI interaction libraries that aid the construction of programs that interact with UIs.

3.2.1 Test Execution

Previously, in Sec. 3.1.1, we outlined the general approach to executing automated tests, i.e. test programs. The bottom line of this approach is that the parts of the SUT are executed as part of the test program, rather than as a separate, independently executed process. This approach is not suitable for acceptance testing because the objective here is to interact with the SUT as a whole, rather than selected sections of its code. When automating acceptance tests, the testing program must therefore interact with an independently executing SUT. Subsequently, interactions with the SUT can only occur on intended channels, such as UI interactions, in order to only test feasible scenarios.

3.2.2 Bounds of Acceptance Testing

This subsection discusses the responsibilities and capabilities of acceptance testing. According to Melnik \cite{44}, acceptance testing intends to demonstrate that the SUT has all the required capabilities. This single responsibility assigned to acceptance testing clearly outlines the approach that acceptance testing should take: By exclusively focusing on demonstrating the correctness of each required feature, acceptance testing is – in essence – a flavor of happy-path testing.

Despite the brief list of responsibilities that acceptance testing has, its capabilities range beyond just determining whether or not the SUT performs the required tasks correctly. With a test failing, not just the plain failure is recorded, but also the reason for it. Such a reason will reflect part of the intrinsic testing model used to construct the test, as the testing model determines what behavior classifies as faulty. Given the relation between behavioral requirements and the testing model that follows from it, the information contained in a test’s
failure will provide at least some insight into the failure’s origin. Such a clue hinting at the possible origin of a failure or bug may be of limited value when considered in isolation. However, in conjunction with information revealed by other tests’ results, experienced testers will be able to deduce far more information from a test failure. It is therefore important that the entire test suite of the SUT be designed carefully, and implemented extensively. Amongst others, this includes unit testing, integration testing and acceptance testing.
Chapter 4

Distributed Systems

This chapter discusses the characteristics of distributed software systems, and the additional considerations required to test them. First, Sec. 4.1 discusses the characteristics of traditional sequential programs, after which Sec. 4.2 introduces the concept of concurrency. It is this section where the notion of a distributed system is explained as well. Next, Sec. 4.3 investigates testing distributed systems, after which Sec. 4.4 will present fundamental building blocks required for engineering such tests.

4.1 Sequential Systems

Traditionally, the simplest of programs are sequential systems: They execute a series of operations in the precise order that they are specified in. Most commonly, the order in which operations are denoted in program code determines this execution order. Specifically in C# code, a statement $o$ that is recorded before statement $p$ in the same scope of a C# source code file will be executed before $p$. Thus, the sequential nature of source code syntax translates directly to the sequential nature of behavior specified using such a syntax.

4.2 Concurrent Systems

In C# 4.0, there is no syntactical construct that facilitates the specification of concurrent behavior. Therefore, this section will provide the reader with background information on concurrency, which will enable us to create such a syntactic facility with precisely defined semantics.

The Oxford English Dictionary (OED) defines “concurrent” as follows [59]

Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; […]

Appropriately, concurrent systems are systems where multiple things happen at the same time. A rudimentary example of such a system is a computer program where the user triggers the execution of two different computations, and gets informed of their results only
4. **Distributed Systems**

Once both computations have completed (Fig. 4.1). This example exhibits the main challenge of concurrency: There is no total order assigned to concurrent operations. In the example above, both triggered operations, say \( o \) and \( p \), could be executed sequentially, in the order \( o \rightarrow p \), in the order \( p \rightarrow o \), or truly in parallel, denoted by \( o \parallel p \). The key here, is to observe that for the specification of the program it does not matter when exactly \( o \) and \( p \) are executed, nor that the program’s user can make any claims about the (lack of) ordering in the execution of \( o \) and \( p \).

This temporal arrangement of operations can be expressed using the **happened-before (HB) relation** [39, 50]. Intuitively, the **HB relation** (denoted by \( a \rightarrow b \)) states that event \( a \) happened before event \( b \) if they are part of the same sequential process, or if the events form a pair of corresponding message send- and message receive-events on concurrent processes. Figure 4.2 illustrates the **HB relation** using an example. From the example we can also observe that the **HB relation** defines a partial order on the set of events \( E \). Considering e.g. only events \( a, b, d, e \) and \( f \), we can merely state \( a \rightarrow b, d \rightarrow e, e \rightarrow f, \) and \( b \rightarrow f \). However, we cannot assume any order of the events \( d \) and \( e \) with respect to \( a \). It is this concept of a partial order that prohibits us from making assumptions about the concrete execution order of the operations \( o \) and \( p \) in the previous example.

4.2.1 **Distributed Systems**

A special case of concurrent systems are distributed systems. The key difference is that distributed systems are designed to run on multiple machines. By contrast, concurrent systems that run on a single machine are called parallel systems. The theoretical foundation of distributed systems does not diverge from what we described in the previous section on concurrent systems. The same properties that follow from the **HB relation** in concurrent systems also define what is conceptually feasible in a distributed system. Furthermore, in theory any concurrent system can be expressed as a distributed system, and vice versa.
4.3 Testing Distributed Systems

Since in this work we are interested in acceptance testing, we will approach testing distributed systems from this very direction. Most importantly, this means that we will treat the distributed SUT as a black box, of which we know only that it spans multiple machines at runtime. Recalling Sec. 3.2.1 on executing acceptance tests, we apply a similar method to reasoning about testing distributed systems. By building a distributed testing system around (parts of) the distributed SUT \[^\text{66}\] , and subsequently interacting with the SUT on intended channels, the test program is structurally separated from the SUT. This approach guarantees the SUT is tested as a black box that functions on its own. Such a distributed test system then facilitates the installation of test probes for observing and stimulating any part of the distributed SUT. Test methods can be implemented using such test probes as interaction primitives, rather than directly invoking the SUT’s subsystems as part of the test program.

4.4 Engineering Distributed Acceptance Tests

Engineering tests around a running distributed system inherently means engineering a separate distributed test system. Since traditional test programs are neither distributed, nor concurrent, we need to revisit how test programs are engineered. For this, we need to gain an understanding of the traits of distributed systems that are required to construct such tests.

While investigating the business case at hand — specifically the NG application — we found that all acceptance tests are sequences of instructions corresponding to the use case being tested. Within such a sequence, instructions are e.g. stimulating the SUT and receiving output from the SUT. Other instructions used are performing general purpose computations, such as deciding whether the SUT’s output conforms to its specification. So far, this model does not differ from traditional test programs, and we might be inclined to think that such sequences defining a one-dimensional chain of HB relations are sufficient for creating such tests. However, for executing distributed acceptance tests, we identified two additional concepts that were added to the semantics of instructions: Concurrency and execution targeting.

4.4.1 Concurrency

Some of the instructions used in manual distributed acceptance test scenarios of NG are in fact composed of multiple instructions that can be executed at the same time. An example of this might be the instruction “Verify that machines A and B are in states S\(_A\) and S\(_B\), respectively.” Recalling the HB relation, this composite instruction does not specify any order in which the two machines’ states ought to be verified, which gives us the freedom to choose any particular execution order, or to truly execute these instructions in parallel. Preceding and subsequent instructions however, must be completed strictly before, and started strictly after this composite instruction, respectively.

From this example, we see that both instructions are partially ordered, following a richer set of HB relations than a plain sequence of instructions would. Visualized by Fig. 4.3, two concurrent instructions or events \(a, b\) occurring between events *Start* and *End* define
4. DISTRIBUTED SYSTEMS

![Diagram of HB relation](image)

Figure 4.3: Illustration of the HB relation of the simplest possible concurrent system. \(a, b\) constitute the set of events \(E\). \(Start\) and \(End\) events have been added for clarifying the HB relation.

multiple [HB relations] following \(Start\), and multiple [HB relations] preceding \(End\). When automating tests that contain such parallel composite instructions, we need a means for expressing this parallelism, such that the concurrent intent is appropriately captured by the test program’s code.

4.4.2 Execution Targeting

In the example instruction from the previous subsection, we identify the second formalism required for engineering distributed acceptance tests: Execution targets. An execution target is an identifier for a machine in a distributed system. In the example above, the concurrently composed instructions each target either "machine A" or "machine B".

In case of a conventional local system, each instruction of that system is executed on the system’s own runtime environment, i.e. each instruction is targeted at the local machine. Since this is a given in most major programming paradigms, the concept of execution targets conventionally remains a quiet implication. It becomes explicitly visible in the context of Remote Procedure Calls (RPCs). Here, remote services exposing methods for execution by clients can only be invoked if the client can distinguish the remote service from itself. Being able to make this distinction demands that the client can obtain an identifier for the remote service it wants to invoke methods on.

For automating distributed tests, we need a mechanism for delegating each individual operation to a specific execution target. This mechanism should not interfere with the execution order of any operations, such that HB relations remain intact.
Chapter 5

The entirety of this chapter dedicated to developing a notation capable of expressing the semantics that Ch. 4 calls for. Section 5.1 explains and motivates the requirements we impose upon the notation we aim to develop. Section 5.2 will introduce the semantic constituents that this notation should capture. From the concepts presented, we proceed to describe an implementation that provides these semantic primitives in a reusable form in Sec. 5.3. An evaluating discussion follows in Sec. 5.4.

5.1 Notation Requirements

Previously, in Sec. 2.3.1 we announced that developing an answer to R.Q. 2.3 should yield a notation capable of expressing interactions with distributed systems, specifically for the purpose of constructing test programs for distributed systems. From the information presented in Sec. 4.4, we conclude that this notation will therefor facilitate the construction of distributed systems itself. Returning to the strategy presented in Sec. 2.3.1, we further demand that proper use of this notation can never result in a distributed system that is subject to concurrency bugs, such as deadlocks. Altogether, we strive to present a notation that encapsulates manageable semantics, which do not demand a high grade of domain-specific knowledge from its users. Finally, the to be developed notation has to fit into the development toolchain that is used in the development effort of the business case at hand, which is C# 4.0 and .NET 4.0.

In the following subsections details regarding the requirements for both expressing distributed behavior (see Sec. 5.1.1) and error prevention (see Sec. 5.1.2) will be revealed. Since the third requirement, ease-of-use, is orthogonal to any requirements for the expressive power of the notation, and since it is hard to formalize, we will not expand upon it further.

5.1.1 Distributed Behavior

Remembering Sec. 4.4, we claim that expressing distributed behavior requires two extensions to the means C# provides for specifying behavior. The first being a matter of communicating a richer set of HB relations among operations to the runtime system. In a traditional
5. **DistrEx**

sequential program, all operations are arranged by a one-dimensional chain of HB relations which follows directly from the order in which operations are denoted in C# code. Our means of communicating extended HB relations, capable of expressing parallelism, should capture semantics for synchronized data in- and output, at the logical beginning and end of parallel behavior.

The second functional requirement for expressing distributed behavior we recall from Sec. 4.4 is the concept of an execution target. The semantics of our notation must envelop this as well, allowing users to explicitly control on which machine an operation is to be executed at runtime.

### 5.1.2 Error Prevention

When constructing a concurrent or distributed program, conventional mechanisms for achieving this goal require their users to take great care in order to avoid introducing faults. While it is arguably a software engineer’s duty to master such skills, manually applying programming patterns that promise error-free systems just as much is not. The reason why the use of such traditional mechanisms poses a risk of faults lies in the fact that their correct use is not guaranteed by the structure of the programs in which they occur. I.e. static analysis by e.g. a syntax checker cannot reveal incorrect use of a mechanism for concurrency.

From this observation we formulate the sole requirement regarding our notation’s ability to express correct concurrent and distributed programs: The notation should be capable of expressing correct concurrent and distributed behavior only. By extension this means that it should not be possible for a syntactically correct program — written using our notation — to exist, while containing faults regarding its concurrent or distributed behavior.

### 5.2 Semantic Primitives

In this section we will introduce semantic primitives, i.e. building blocks, that allow us to reason about constructing programs. We base our discussion on Church’s [λ-calculus][23], but we will communicate our ideas using a different syntax than Church’s. Ultimately, we seek to deliver a declarative syntax definition for a context-free grammar G defining the notation to be developed, as a result of cleverly choosing this communicating syntax. As context-free grammars are suitable for adaptation into language processing programs, such as syntax checkers, we hope to profit from this effort in Sec. 5.3 where G is implemented.

When defining G, we will focus mostly on providing context-free production rules, which will be provided in Backus-Naur Form (BNF)[5]. Accompanied by syntax tree construction rules and corresponding abstract syntax tree representations, these context-free production rules allow us to compose visual representations of programs denoted in G.

We will start in Sec. 5.2.1 by discussing the most basic building block of any program: Operations. Section 5.2.2 then continues to explain the works of sequential composition, the next basic requirement for constructing arbitrarily complex sequential programs. From this foundation, we proceed to add parallel composition in Sec. 5.2.3 and instruction targeting in Sec. 5.2.4. Finally, we will introduce two-ply operations, a useful concept that helps to avoid race conditions in parallel systems (Sec. 5.2.5).
5.2.1 Operations

An operation is the most fundamental element of a program. Different names for same concept are ‘computation’, ‘function’ and ‘instruction’. An operation $F_i$ is the process of transforming an input value (argument) $a_i$ of type $A_i$ to an output (result) $r_i$ of type $R_i$. In formal mathematics, this is denoted by $F_i: A_i \rightarrow R_i$. Users who wish to employ the operation $F_i$ are merely interested in its capabilities, i.e. in its input an output types, $A_i$ and $R_i$ respectively. From the perspective of a user, an operation $F_i$ can therefor be sufficiently defined as a pair of argument and result types: $F_i := \langle A_i, R_i \rangle$. In the future, however, we will not include operation’s argument or result types in our grammar $G$, since we will be leveraging an existing type system when implementing $G$. The discussion of these types merely aids the reader by putting the concept of an operation into the right context. Note that we are not considering the internal characteristics of the operation $F_i$, such as what its functional behavior is, or how exhibiting that behavior might be accomplished.

5.2.2 Sequential Composition

With a formal definition of operations at our fingertips, we now proceed to define the next fundamental building block required to express arbitrary sequential programs. This building block is the glue between operations that allows the composition of complex operations from simple operations. By ‘sequential composition’ we mean using two operations $F_i, F_{i+1}$ in order, i.e. $F_i$ is applied to argument $a_i$ first, after which $F_{i+1}$ is applied to $F_i$’s result $r_i$, which will eventually yield result $r_{i+1}$. The reader may already have identified a constraint which this composition puts on the second operation’s argument type: $F_{i+1}$’s argument type $A_{i+1}$ must match $F_i$’s result type $R_i$ ($R_i = A_{i+1}$).

Sequential composition of two operations $F_i, F_{i+1}$ can be denoted as nested function application on the initial argument $a_i$: $F_{i+1}(F_i(a_i))$. We will divert from this convention in order to better reveal the $\text{HB relation}$ that follows from sequential composition. We propose the notation $F_i \rightarrow F_{i+1}$ for sequential composition. Purposely chosen, this notation is equivalent the notation for the $\text{HB relation}$ used in Sec. 4.2 if we consider the terms ‘event’ and ‘operation’ to be equivalents of each other.

The sequential composition $F_i \rightarrow F_{i+1}$ of two operations $F_i, F_{i+1}$ conceptually yields an operation $F_j$ of its own. For its users, this newly created operation $F_j$ is defined by its argument and result type: $F_j := \langle A_j, R_{j+1} \rangle$, or equivalently $F_j := \langle A_j, R_j \rangle$ where $A_j = A_i$, $R_j = R_{i+1}$. Such a composed operation $F_j$ can itself be used in a composition with operation $F_{i+2}$, in any way the user chooses to: $F_j \rightarrow F_{i+2}$ or $F_{i+2} \rightarrow F_j$. These nested compositions can be written in full length as $(F_i \rightarrow F_{i+1}) \rightarrow F_{i+2}$ and $F_{i+2} \rightarrow (F_i \rightarrow F_{i+1})$, respectively. Had a different order of nesting compositions been chosen, the sequence $(F_i \rightarrow F_{i+1}) \rightarrow F_{i+2}$ would have been written as the equivalent $F_i \rightarrow (F_{i+1} \rightarrow F_{i+2})$. From this observation we conclude that the sequential composition operator ($\rightarrow$) is associative, which means that multiple uses of the operator in one sentence do not require parentheses, e.g. $F_i \rightarrow F_{i+1} \rightarrow F_{i+2}$.

The allowed nesting of sequentially composed operations enables us to formulate a production rule P.R. for the context-free grammar $G$ that will be the formal basis for the notation being developed. In order to define proper production rules, we also introduce the
concept of an elementary operation, which is an operation that is not composed of other operations (P.R. 1). In our grammar \( G \), elementary operations will be treated as terminal symbols. These production rules can guide the creation of syntax trees, using the respective corresponding syntax tree construction rules in Fig. 5.1 and Fig. 5.2.

**Production Rule 1** \( \text{Operation} \rightarrow \text{Elementary Operation} \)

**Production Rule 2** \( \text{Operation} \rightarrow \text{Operation} \rightarrow \text{Operation} \)

**Example**

We will now demonstrate by example how P.R. 1 and P.R. 2 are to be used for constructing abstract syntax trees. We will start by choosing the sentence \( F_i \rightarrow F_{i+1} \rightarrow F_{i+2} \) as the basis for this example. Following the syntax tree construction rules in Fig. 5.1(b) and Fig. 5.2(b), we construct the syntax tree in Fig. 5.3.
For the same sentence, we can construct an execution graph. Such a graph visually communicates the execution order of operations. Figure 5.4 presents the execution graph for the sentence above.

5.2.3 Parallel Composition

As we recall from Sec. 5.1.1, parallelism can be achieved by defining HB relations that are not just sequential chains, such as those resulting from sequential composition, which we have seen so far.

Figure 4.3 illustrates the HB relations involved in a rudimentary parallel system. From this illustration we observe that starting parallelism between two events $a$ and $b$ involves HB relations such that $s \rightarrow a$ and $s \rightarrow b$, for some event $s$ that precedes both $a$ and $b$. Symmetrically, we can observe that terminating parallelism between two events $a$ and $b$ involves HB relations such that $a \rightarrow e$ and $b \rightarrow e$, for some event $e$ that follows both $a$ and $b$. This symmetric model of concurrent events, which begins and ends with non-concurrency, allows us to formulate a model for concurrent operations analog to sequential operations.

Previously, in Sec. 5.2.2, we have described an operation $F_i$ as the pair of its argument and result types: $F_i := \langle A_i, R_i \rangle$. Furthermore, we have declared sequentially composed operations $F_i, F_{i+1}$ to be operations themselves, plugging in this definition of argument and result types: $F_j := \langle A_i, R_{i+1} \rangle$. A similar pattern can be followed for defining parallely composed operations. First, we chose the symbol $\parallel$ as the binary operator for denoting parallelism: $F_k \parallel F_l$ means operations $F_k, F_l$ are to be executed in parallel. Since we want $F_k \parallel F_l$ to be an operation of its own, named $F_j$, we define it in terms of its argument and result types: $F_j := \langle A_k, \langle R_k, R_l \rangle \rangle$, where $A_k = A_l$. Note that the result type of the parallel composition $F_j$ is a vector $\langle R_k, R_l \rangle$ of both inner result types.

To add the parallel composition operator $\parallel$ to our grammar $G$, we introduce P.R. 3 and the corresponding syntax tree construction rule in Fig. 5.5. This operator is associative as well, and it has a higher priority than the sequential composition ($\rightarrow$) operator.

Production Rule 3  Operation $\rightarrow$ Operation “$\parallel$” Operation

Example

We will now demonstrate by example how P.R. 3 is to be used for constructing abstract syntax trees. We will start by choosing the sentence $F_i \rightarrow F_k \parallel F_i \rightarrow F_{i+1}$ as the basis for this example. Following the syntax tree construction rules in Fig. 5.1(b), Fig. 5.2(b) and Fig. 5.2(b), we construct the syntax tree in Fig. 5.6. The execution graph for the same sentence is provided in Fig. 5.7.
5. **DISTReX**

![Syntax tree construction rule for PR. 3](image)

![Syntax tree abstraction for PR. 3](image)

Figure 5.5: Production Rule 3 syntax tree and abstract syntax tree representation.

![Syntax tree](image)

![Abstract syntax tree](image)

Figure 5.6: Syntax tree for sentence $F_i \rightarrow F_k \parallel F_l \rightarrow F_{i+1}$.

![Execution graph](image)

Figure 5.7: Execution graph for sentence $F_i \rightarrow F_k \parallel F_l \rightarrow F_{i+1}$. Labels on transitions reveal argument and result types for operations.

### 5.2.4 Instruction Targeting

The last remaining component required for constructing distributed systems is execution targeting, as discussed in Sec. 4.4.2. To find the right place for introducing this concept to our grammar $G$, we first reiterate the responsibilities of semantic concepts we have added to $G$ so far.

Sequential and parallel composition of operations are merely a matter of synchronization. The execution target of synchronization behavior does not matter, since it only involves a composed operation’s internal execution planning. Elementary operations on the other hand are merely wrapped in composed operations, but strictly do not have any knowledge of the composed operation they are part of. At the same time, elementary operations are the primitives that can interact with the world outside of a composed operation, e.g. to interact with a SUT.

**Production Rule 1**  
$\text{Operation} \rightarrow \text{Elementary Operation}$
If we recall P.R. 1, we see that it is this rule that connects the world of elementary instructions with the world of composed operations. We therefore retract P.R. 1 and replace it with the revised P.R. 4, which includes semantics for an execution target $X_p$. In order to define P.R. 4, we introduce the $·$ symbol as the operator for coupling operations to an execution target: $X_p · F_i$ means that operation $F_i$ is to be executed by execution target $X_p$. Like elementary operations, execution targets will be treated as terminal symbols. Figure 5.8 shows the syntax tree construction rule corresponding to P.R. 4.

**Production Rule 4**

$\text{Operation} \rightarrow \text{Execution Target} \; \cdot \; \text{Elementary Operation}$

Following this revised rule, previous examples using the grammar $G$ should be rewritten to include execution targeting: $X_p · F_i \rightarrow X_p · F_{i+1}$ and $X_p · F_k \parallel X_q · F_l$ show the use of the parallel composition operator in revised examples.

**Example**

We will now demonstrate by example how P.R. 4 is to be used for constructing abstract syntax trees. We will use the example sentence from Sec. 5.2.3 as the basis for this example and rewrite it to incorporate use of P.R. 4: $X_p · F_i \rightarrow X_p · F_{i+1}$ and $X_q · F_l \parallel X_q · F_i \rightarrow X_q · F_{i+1}$. Following the syntax tree construction rules in Fig. 5.2(b) and Fig. 5.8(b), we construct the syntax tree in Fig. 5.9. The execution graph for the same sentence is provided in Fig. 5.10.

**5.2.5 Two-Ply Operations**

A common pattern in automated acceptance testing of NG is checking whether a certain input causes a certain output. In practice, this involves installing an event listener of some sorts, that will make sure the expected event is observed once it occurs. After the listener has been installed, the test program can send a stimulus to the SUT that should cause the expected event to occur. Once the event has been observed, it is reported to the test program and the event listener can be uninstalled. In order to avoid race conditions, the event listener should be installed strictly before the operation sending the stimulus to the SUT is executed, and should not be uninstalled until strictly after. So far, the operations we have defined can only ever be in one of three states: Not yet executed (pre), currently executing (live), and executed (post). It is only during the live state that an operation may use any resources, such as event listeners.
5. **DISTREX**

![Syntax Tree](image)

**(a) Syntax tree.**

![Abstract Syntax Tree](image)

**(b) Abstract syntax tree.**

Figure 5.9: Syntax tree for sentence $X_p \cdot F_i \rightarrow X_p \cdot F_k \parallel X_q \cdot F_l \rightarrow X_q \cdot F_{i+1}$. $X_p$ is represented by blue, and $X_q$ is represented by orange.

![Execution Graph](image)

Figure 5.10: Execution graph for sentence $X_p \cdot F_i \rightarrow X_p \cdot F_k \parallel X_q \cdot F_l \rightarrow X_q \cdot F_{i+1}$. Labels on transitions reveal argument and result types for operations. $X_p$ is represented by blue, and $X_q$ is represented by orange.
The grammar $G$ we have defined so far thus does not allow us to compose an operation that implements the scenario defined above. To overcome this, we introduce a new kind of operation, named two-ply operation, that has five states: Not yet executed (pre), currently executing the beginning (live1), currently executing unsupervised (unsupervised), currently executing the end (live2) and executed (post). The name is motivated by the fact that the two-ply operation decides when to transition from live1 to unsupervised, while it has no control over other state transitions. The point at which this transition occurs splits the two-ply operation in half, dividing it into separate smaller operations.

When a two-ply operation leaves the live1 state, it has completed the first part of its work, after which it returns control to the runtime system that coordinates operations’ execution. Depending on the operation composition, the runtime system can now execute any other operation while the two-ply operation remains in execution on its target. The now unsupervised operation will transition to the live2 state once the runtime system signals its interest in the result of the second half of the two-ply operation. From that point on, the two-ply operation acts like a regular operation, terminating once it has completed. Figure 5.11 illustrates the states of regular operations and two-ply operations. The illustration of two-ply operations in Fig. 5.11(b) depicts the example scenario above.

As a whole, a two-ply operation $T_i$ has characteristics similar to a regular operation. It has an argument type $A_i$ and a result type $R_i$, by which we can define the two-ply operation:

$$T_i := \langle A_i, R_i \rangle$$

However, a two-ply operation also has a first part $T'_i$ and a second part $T''_i$, which correspond to its live1 and live2 states. If we wish to use $T'_i, T''_i$ as separate operations in compositions, we can define them like regular operations, using their argument and result types: $T'_i := \langle A_i, I \rangle$ and $T''_i := \langle I, R_i \rangle$. Here, $I$ is the type of an identifier. Executing the first part of a two-ply instruction $T_i$ yields an identifier $i_i$ of type $I$, by which the unsupervised operation $T_i$ can later be recovered. Executing the second part of $T_i$ requires $i_i$ to be the supplied argument to $T''_i$, after which a result of type $R_i$ will be returned. Using the grammar $G$ developed so far, executing $T_i$ as a whole involves a sequential composition of $T'_i$ and $T''_i$:

$$X_p \cdot T'_i \rightarrow X_p \cdot T''_i.$$
5. DistriEx

Operations, two-ply operations are always considered to be terminal symbols.

Production Rule 5  $\text{Operation} \rightarrow \text{Execution Target} \cdot \cdot \cdot \text{Two-Ply Operation}$

Production Rule 6  $\text{Operation} \rightarrow \text{Execution Target} : \cdot \cdot \cdot \text{Two-Ply Operation}$

The example scenario from the beginning of this subsection, which is also depicted in Fig. 5.11(b), can now be expressed as the following sentence: $X_p \cdot T_i \rightarrow X_p \cdot F_{i+1} \rightarrow X_p \cdot F_{i+2} \rightarrow X_p : T_i$.

Example

We will now demonstrate by example how P.R. 5 and P.R. 6 are to be used for constructing abstract syntax trees. We will use a similar example sentence as we did in Fig. 5.11(b), with the addition of a second execution target for one operation: $X_p \cdot T_i \rightarrow X_q \cdot F_{i+1} \rightarrow X_p \cdot F_{i+2} \rightarrow X_p : T_i$. Following the syntax tree construction rules in Fig. 5.2(b), Fig. 5.2(b), Fig. 5.8(b), Fig. 5.12(b) and Fig. 5.13(b) we construct the syntax tree in Fig. 5.14. The execution graph for the same sentence is provided in Fig. 5.15.

5.3 Implementation

The grammar $G$ which primarily consists of production rules P.R. 2 through P.R. 6 must be applicable to the business case at hand, and aid the construction of distributed acceptance tests of the NG application. Since NG is mostly written using the C# language, it is sensible to integrate $G$ into C#. We have achieved this by creating a fluent Application Programming...
Interface (API) that reads much like a Domain Specific Language (DSL) would. This API is available as a library named DistrEx.

In Sec. 5.3.1 we will first map all operators in $G$ to their corresponding DistrEx API function. Section 5.3.2 and Sec. 5.3.3 will then review challenges that arise from remotely executing operations, and propose solutions to those challenges. Details concerning the DistrEx runtime system are presented in Sec. 5.3.4. Finally, Sec. 5.3.5 revisits the examples sentences from Sec. 5.2 and demonstrate how they can be implemented using the DistrEx API.

The DistrEx source code is available at https://github.com/derabbink/DistrEx
5. DistrEx

5.3.1 The DistrEx API

Elementary Operations

The first construct of our grammar $G$ that we will translate into an API function is the elementary operation. Since C# supports the notion of delegate types — which are types describing a function’s signature — we can simply define elementary operations as C# functions that match a certain signature. Listing 5.1 shows the C# type definitions for both elementary operations (Instruction), and elementary two-ply operations (TwoPartInstruction). Both these definitions follow the scheme we proposed in Sec. 5.2, where operations are described by their argument and result types (TArgument and TResult).

Compared to an Instruction, a TwoPartInstruction has an additional function argument (Action reportCompletedPart1), which contains a delegate that the two-ply operation can invoke to signal completion of its live1 state. Furthermore, both type definitions demand two additional function arguments: CancellationToken cancellationToken and Action reportProgress. These will be explained in Sec. 5.3.2 and Sec. 5.3.3, but for the time being we will disregard them.

```csharp
delegate TResult DistrEx.Common.Instruction<TArgument, TResult>(CancellationToken cancellationToken, Action reportProgress, TArgument argument);
delegate TResult DistrEx.Common.TwoPartInstruction<TArgument, TResult>(CancellationToken cancellationToken, Action reportProgress, Action reportCompletedPart1, TArgument argument);
```

Listing 5.1: Elementary operations’ type definitions.

The user of the DistrEx API who wishes to specify operations and two-ply operations can now do so by assigning member functions and anonymous functions to appropriately typed variables, as Ls. 5.2 demonstrates. It is worth mentioning that in its current implementation, DistrEx only supports statically defined operation delegates. These are either statically defined methods, or anonymous delegates that do not contain closed-over objects, as these will be compiled to member functions of derived closure types.

```csharp
// store delegates in variables
Instruction<double, int> instrCeil = Ceil;
TwoPartInstruction<double, int> instrFloor = Floor;

static int Ceil(CancellationToken ct, Action rp, double arg) {
    // ignore ct and rp
    return Convert.ToInt32(Math.Ceil(arg));
}
static int Floor(CancellationToken ct, Action rp, Action plComplete, double arg) {
    // ignore ct and rp
```
double modulo = arg % 1;
// signal that part 1 is complete
p1Complete();
// continue with part 2
return Convert.ToInt32(arg - modulo);
}

Listing 5.2: Using elementary operation types.

Execution Targeting

The type definitions from the previous subsection allow us to introduce variables for elementary operations. By themselves, such elementary operations can only be executed locally, as part of the conventional execution flow of a program. In order to let operations be executed on any given execution target, we must first create a means for distinguishing different execution targets. Similar to the approach in Sec. 5.2.4, we do this by using identities that represent execution targets.

In DistrEx, the execution target identity is embodied by the abstract TargetSpec class, of which two concrete implementations exist. The first implementation (OnCoordinator) represents the local execution target, while the second implementation (OnWorker) can be instantiated to represent a remote target. Listing 5.3 reveals the fully qualified class names for all three types.

abstract class DistrEx.Coordinator.Interface.TargetSpec {
    // ...
}
class DistrEx.Coordinator.TargetSpecs.OnCoordinator :
    DistrEx.Coordinator.Interface.TargetSpec {
        // ...
}
class DistrEx.Coordinator.TargetSpecs.OnWorker :
    DistrEx.Coordinator.Interface.TargetSpec {
        // ...
}

Listing 5.3: Execution target type definitions.

Within the TargetSpec class, the execution targeting operators · and : are implemented as member functions. The · operator for both regular and two-ply operations is implemented by two overloading methods named Do, while the : operator is implemented by a member function named GetAsyncResult. Both Do and GetAsyncResult functions return an altered representation (TargetedInstruction) of the elementary operation they operate on, which contains the execution target’s identity. A TargetedInstruction can then be invoked on the designated target by using its Invoke member function. Listing 5.4 exhibits the type definitions involved in instruction targeting.

using System;
using DistrEx.Common;
using DistrEx.Coordinator.Interface.TargetedInstructions;

abstract class DistrEx.Coordinator.Interface.TargetSpec
{
    TargetedInstruction<TArgument, TResult> Do<TArgument, TResult>(Instruction<TArgument, TResult> instruction) {
        // ...
    }
    TargetedInstruction<TArgument, Guid> Do<TArgument, TResult>(TwoPartInstruction<TArgument, TResult> asyncInstruction) {
        // ...
    }
    TargetedInstruction<Guid, TResult> GetAsyncResult<TResult>() {
        // ...
    }
    // ...
}

abstract class
TargetedInstruction<TArgument, TResult> {
    abstract Future<TResult> Invoke(TArgument argument);
    // ...
}

Listing 5.4: Type definitions related to targeted operations.

As Ls. 5.4 shows, the DistrEx implementation follows the notion from Sec. 5.2.4, which states that two-ply operations can be represented as two separate operations, of which the first returns, and the second expects an identifier (of type Guid).

As the reader may have noticed, the return type of a TargetedInstruction’s Invoke method has changed from the elementary operation’s TResult to Future<TResult>. This type represents a value that may not be available, due to an asynchronous computation that has not yet completed. Later on, when explaining DistrEx’s implementation of parallel composition, we will explore this concept further.

Sequential Composition

Previously, we argued that the coordination of individual operation’s execution is not any particular execution target’s responsibility. It is only important that operations are executed in the right order, on their intended execution targets. To separate the concern of execution coordination from actual execution, we have introduced the Coordinator class, with the static Do factory method. This method transforms a TargetedInstruction instance into
a CoordinatorInstruction, on which the sequential composition operator \( \rightarrow \) is defined. This operator is implemented by the ThenDo member function, which takes the operator’s right hand operand as an argument of type TargetedInstruction. The ThenDo operation creates a new composite CoordinatorInstruction, which can be used in further compositions. The types involved are shown in Ls. 5.5.

```
using DistrEx.Coordinator.Interface.TargetedInstructions;

static partial class DistrEx.Coordinator.Coordinator {
    static CoordinatorInstruction<TArgument, TResult> Do<TArgument, TResult>(TargetedInstruction<TArgument, TResult> targetedInstruction) {
        // ...
    }
    // ...
}

class DistrEx.Coordinator.TargetedInstructions.CoordinatorInstruction<TArgument, TResult> : TargetedInstruction<TArgument, TResult> {
    CoordinatorInstruction<TArgument, TNextResult> ThenDo<TNextResult>(TargetedInstruction<TResult, TNextResult> nextInstruction) {
        // ...
    }
    // ...
}
```

Listing 5.5: Type definitions related to coordinated operations.

Careful examination of the ThenDo method’s signature reveals that the requirement for matching result and argument types \( (R_i = A_{i+1}) \) of two sequentially arranged operations \( F_i, F_{i+1} \) is captured by DistrEx as well.

**Parallel Composition**

Syntactically, the parallel composition operator \( \parallel \) is very similar to the sequential composition operator \( \rightarrow \). Both operate on two operands and yield a composite operation that can be used in further compositions. However, since a in \( \#\) it is not possible to implement infix operators with different priorities as methods of a fluent API we have chosen to implement the \( \parallel \) operator as a prefix operator. The operator is implemented as overloads of the Do and ThenDo methods we have seen before, with an additional argument for the operator’s second operand. This implementation allows us to specify parallel operations as siblings in the syntax tree directly, rather than having to rely on the parser to construct appropriately nested trees. As a result, the DistrEx API allows users to write less convoluted code than an infix implementation would, since operator priorities of a fluent API can only be com-
municated manually, by using parentheses. The function overloads that implement parallel composition of operations are shown in Ls. 5.6

```csharp
using DistrEx.Coordinator.Interface.TargetedInstructions;

static partial class DistrEx.Coordinator.Coordinator {
    static CoordinatorInstruction<TArgument, Tuple<TResult1, TResult2>> Do<TArgument, TResult1, TResult2>(TargetedInstruction<TArgument, TResult1> targetedInstruction1, TargetedInstruction<TArgument, TResult2> targetedInstruction2) {
        // ...
    }
    static CoordinatorInstruction<TArgument, Tuple<TNextResult1, TNextResult2>> ThenDo<TArgument, TIntermediateResult, TNextResult1, TNextResult2>(this CoordinatorInstruction<TArgument, TIntermediateResult> instruction, TargetedInstruction<TIntermediateResult, TNextResult1> nextInstruction1, TargetedInstruction<TIntermediateResult, TNextResult2> nextInstruction2) {
        // ...
    }
    // ...
}
```

Listing 5.6: Function declarations for parallel composition.

In line with the formal definition of the $\parallel$ operator in Sec. 5.2.3 the argument types for both composed operations $F_k, F_l$ must match ($A_k = A_l$), and the result type of the composite operation is a vector containing both inner results ($\langle R_k, R_l \rangle$).

**Syntactic Sugar**

Since we have implemented the parallel composition operator $\parallel$ as a function that takes both the operator’s operands as arguments, we can easily extend that implementation to accept more operands. That way, nested application of the $\parallel$ operator can be flattened to a single use of the operator. Listing 5.7 shows the corresponding function overloads for the parallel composition of three TargetedInstruction instances. Analogous function overloads are provided by DistrEx for up to seven operands. The benefit of this approach is twofold. Firstly, DistrEx users can write nested parallel compositions more concisely, and secondly,
the \texttt{DistrEx} runtime system can more easily optimize the execution of such compositions if they are flattened.

```csharp
using DistrEx.Coordinator.Interface.TargetedInstructions;

static partial class DistrEx.Coordinator.Coordinator {
    static CoordinatorInstruction<TArgument, Tuple<TResult1, TResult2, TResult3>>
        Do<TArgument, TResult1, TResult2, TResult3>(TargetedInstruction<TArgument, TResult1> targetedInstruction1,
            TargetedInstruction<TArgument, TResult2> targetedInstruction2,
            TargetedInstruction<TArgument, TResult3> targetedInstruction3) {
        // ...
    }

    static CoordinatorInstruction<TArgument, Tuple<TNextResult1, TNextResult2, TNextResult3>>
        ThenDo<TArgument, TIntermediateResult, TNextResult1, TNextResult2, TNextResult3>(
            this CoordinatorInstruction<TArgument, TIntermediateResult> instruction,
            TargetedInstruction<TIntermediateResult, TNextResult1> nextInstruction1,
            TargetedInstruction<TIntermediateResult, TNextResult2> nextInstruction2,
            TargetedInstruction<TIntermediateResult, TNextResult3> nextInstruction3) {
        // ...
    }

    // ...
}
```

Listing 5.7: Function declarations for parallel composition.

### 5.3.2 Quiescence

Recalling Ls.\textsuperscript{5,4} we see that invoking a \texttt{TargetedInstruction} by its \texttt{Invoke} method returns a \texttt{Future} object that represents the operation’s yet to-be-computed result. Such a \texttt{Future} object is returned immediately after calling \texttt{Invoke}, which asynchronously starts the \texttt{TargetedInstruction} on its designated target, without waiting for it to complete. Once the execution is complete, the operations result is reported back to the caller. This leaves the caller, which is the \texttt{DistrEx} coordinator, free to do other work while the started operation is executed on a different part of the system.
Since operation execution happens asynchronously, possibly on a remote target, the operation’s caller has no control over it, once it is started. A common problem with concurrent systems is that any component is at risk of failing at any time. The distributed nature of DistrEx means that a failing execution target will also fail to report any operations’ results back to their callers. In order to recover from it, the system needs to be able to identify failures of its components. Instead of having to wait indefinitely for an operation’s result that will not arrive, we model this failure to report back as an observable event. This event, characterized by the lack of output, is called ‘quiescence’.

DistrEx implements quiescence as an expiring timeout: For each operation that the DistrEx coordinator invokes, it starts a countdown timer. If a countdown expires before the corresponding operation’s result has been reported back, the execution of that operation is considered to have failed. Naturally, there is no reliable way of predicting the execution time for arbitrary operations, which makes it likely that any predefined timeout length will be inappropriate for most situations. For this reason, we have included the reportProgress argument in the type definitions of Instruction and TwoPartInstruction. By invoking the reportProgress delegate, an operation can signal that it has not succumbed to any failure, and that it is still executing properly. The reportProgress argument is provided by the DistrEx runtime system, and its invocation causes the appropriate countdown timer to be reset to its initial value, prolonging the expiration time. Through this argument, an operation’s implementer has the freedom of implementing arbitrarily long-running operations.

5.3.3 Termination

The definitions of DistrEx’s elementary operation delegates foresee a simple lifecycle for their execution: Invocation, execution, and return. Normally, reaching a function’s return statement means a clean, error-free conclusion of its execution. In C#, however, functions can terminate unexpectedly by throwing an exception. In testing, such faults are often provoked deliberately, to ensure the SUT responds predictably to stimuli it cannot process properly. In order to support this testing strategy, exceptions thrown in operations must be thrown such that DistrEx users’ code can process them without interrupting the DistrEx runtime. This holds true even for operations that are executed on a remote target. From this perspective, any termination of an operation, whether expected or unexpected, should be reported to the user equally transparently. The DistrEx runtime provides this functionality by catching all exceptions thrown in operations, and rethrowing them for user’s code to process.

Since composing operations with DistrEx is built on the premise that the result of one operation is the argument to the next operation, execution of a composite operation cannot continue after an exception was thrown. This means that the DistrEx runtime will not execute any sequential successor operations following a failed operation. Moreover, any operations executing in parallel with a thrown exception will be canceled as soon as the exception is detected. For this reason, Instructions and TwoPartInstructions receive a CancellationToken argument, which is used to signal a request for cancellation. It is the user’s duty to implement operations with cooperative cancellation, as the DistrEx runtime does not forcefully end the execution of running operations.
5.3.4 The DistrEx Runtime System

Previous sections have referred to the DistrEx runtime system on numerous occasions. From these mentions, we have only learned that both the coordination and targeted execution of operations are duties of this runtime system. This section will explain how these tasks are accomplished.

A DistrEx runtime system primarily consists of a coordinator, and any number of execution targets. The coordinator is implemented by DistrEx API functions that send service requests to workers and control the execution of composite operations. As such, the coordinator is executed as part of the application that uses the DistrEx API. An execution target is implemented by a service application named ‘DistrEx.Worker.Host’. In the following, instances of this service application will be referred to as ‘workers’.

Execution Service Functions

In a traditional service application, the exposed functionality is limited to a predefined set of methods. Since with DistrEx we want to remotely execute arbitrary operations, which are at the most primitive level defined as C# delegates, we need to expose a higher order function as a service. Such a service function will then be able to execute another function — or delegate — that is provided to it as an argument. While C# allows the creation of such higher order functions in regular code, sending delegates to a remote service is strictly not possible. Since DistrEx implements its service functions using Microsoft’s Windows Communication Foundation (WCF) library, service function arguments and results must be serializable values, which delegates are not.

To this end, we use Reflection to create a serializable identifier for each operation delegate we want to execute remotely. Such an identifier is implemented by the Spec type, of which two sub classes exist that identify regular and two-ply operations. Both sub classes expose a similar method (GetDelegate) that produces a delegate from their specification, which is then invoked by the DistrEx worker service. The definitions of these functions and types are provided in Ls. 5.8

```csharp
using DistrEx.Common;

abstract class DistrEx.Coordinator.Interface.Spec {
  abstract string GetAssemblyQualifiedName();
  abstract string GetMethodName();
  // ...
}

  abstract Instruction<TArgument, TResult> GetDelegate();
  // ...
}
```
abstract class DistrEx.Coordinator.Interface.
AsyncInstructionSpec<TArgument, TResult> : 
DistrEx.Coordinator.Interface.Spec {
    abstract TwoPartInstruction<TArgument, TResult> GetDelegate();
    // ...
}

Listing 5.8: Types for instructions’ reflection identifiers.

On-Demand Deployment

The GetDelegate methods can only transform a delegate’s identifier into an actual delegate, if the code defining the delegate in question is available to the process executing the method. On the side of the coordinator, this is certainly the case, as the code defining any Instructions or TwoPartInstructions will be part of the process that will execute the coordinator through its [API] method calls. A worker, however, is a separate process, which cannot be deployed together with all possible assemblies from which coordinators might want to execute operations.

Therefore, before any operation can be executed on a worker, the code that defines those operations must be deployed to the worker. In C# units of code are organized in assemblies, which are compiled into Dynamic Link Library (DLL) files. Thus, for a worker to execute an operation’s code, it has to have access to the compiled assembly file in which that operation is defined. DistrEx achieves this by sending required assembly files to workers before requesting the execution of operations. To this end, the worker application exposes a second service, capable of receiving assembly files that will be available for use by future operations. Both service interfaces that a DistrEx worker exports are shown in Ls. 5.9

using System.ServiceModel;
using DistrEx.Communication.Contracts.Data;

[ServiceContract (Name = "Executor", Namespace = "http://schemas.fugro/distrex/service/executor",
CallbackContract = typeof (IExecutorCallback))]
interface DistrEx.Communication.Contracts.Service.IExecutor {
    [OperationContract (IsOneWay = true)]
    void Execute (Instruction instruction);
    [OperationContract (IsOneWay = true)]
    void ExecuteAsync (AsyncInstruction asyncInstruction);
    [OperationContract (IsOneWay = true)]
    void GetAsyncResult (GetAsyncResultInstruction getAsyncResultInstruction);
    [OperationContract (IsOneWay = true)]
}
void ClearAsyncResults();
[OperationContract(IsOneWay = true)]
void Cancel(Cancellation cancellation);
}
[ServiceContract(Name = "AssemblyManager", Namespace =
   "http://schemas.fugro/distrex/service/assemblymanager")]
interface DistrEx.Communication.Contracts.Service.IAssemblyManager {
   [OperationContract]
   void AddAssembly(Assembly assembly);
   [OperationContract]
   void Clear();
}

Listing 5.9: Worker service interfaces.

Executing Foreign Code

The DistrEx worker application is deployed as a bundle containing only its own assemblies and those assemblies that are strictly required as dependencies. By extension, it is not aware of any foreign assemblies that contain operations to execute. Reflection can facilitate loading additional assemblies, but the architecture of .NET which eventually handles the assembly loading process, does not allow unloading assemblies which are no longer needed. .NET can, however, unload entire AppDomains. Since every .NET process automatically has a default AppDomain which cannot be unloaded, it is necessary to create a second — temporary — AppDomain to load such assemblies into. Once the loaded foreign assemblies are no longer required, the entire temporary AppDomain can be unloaded.

Traditionally, a .NET application’s code executes all within its default AppDomain. Since AppDomains are separate sections of a .NET process’s memory, in-memory communication such as method invocations between AppDomains is restricted. The only possible communication between AppDomains requires invoking member functions of marshalled objects. Although conceptually similar to exposing a service on a network, this process happens entirely in memory, and is managed by .NET.

Listing 5.10 shows the interface of a marshalled Executor object that facilitates this cross AppDomain communication in DistrEx.

using DistrEx.Plugin;

class DistrEx.Plugin.Executor : MarshalByRefObject {
  SerializedResult Execute(ExecutorCallback callback,
   string assemblyQualifiedName, string methodName,
   string argumentTypeName, string serializedArgument) {
      // ...
}
Operation Execution

Figure 5.16: Steps of executing a DistrEx operation. Indicates a DistrEx component. Indicates a component implemented by DistrEx’s user. Table 5.1 describes each numbered step.

```csharp
SerializerResult ExecuteTwoStep(ExecutorCallback callback, ExecutorCallback completedStep1,
    string assemblyQualifiedName, string methodName,
    string argumentTypeName, string serializedArgument) {
    // ...
    }
    // ...
```

Listing 5.10: Definition of marshalled execution facilitator.

DistrEx workers create a new AppDomain before each operation they execute, and unload it immediately after its execution is complete. Operation arguments and results, progress reports and cancellation requests travel from the coordinator process, over the network, to the worker’s exposed services, and from there across AppDomain boundaries to the marshalled Executor object. Information is serialized at the beginning of this chain of communication, and only deserialized at the other end, such that only generic binary data streams are transported between both ends. Figure 5.16 accompanied by Tbl. 5.1 schematically explains the steps involved in executing operations on a worker.

5.3.5 Examples

Since DistrEx executes operations asynchronously, this behavior is also exposed by the DistrEx API. This way, API users are free to benefit from asynchronous execution of operations.
Table 5.1: Steps of executing a DistrEx operation, as depicted in Fig. 5.16

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Invoking coordinator</td>
</tr>
<tr>
<td>2</td>
<td>Deploying assemblies</td>
</tr>
<tr>
<td>3</td>
<td>Requesting execution by executor service</td>
</tr>
<tr>
<td>4</td>
<td>Invoking executor in temporary AppDomain</td>
</tr>
<tr>
<td>5</td>
<td>Invoking operation delegate</td>
</tr>
<tr>
<td>6</td>
<td>Returning result</td>
</tr>
<tr>
<td>7</td>
<td>Reporting result to executor service</td>
</tr>
<tr>
<td>8</td>
<td>Reporting result to coordinator</td>
</tr>
<tr>
<td>9</td>
<td>Returning result to user’s program</td>
</tr>
</tbody>
</table>

if needed. Invoking a DistrEx operation involves calling an overload of the Coordinator’s Do method, which takes a TargetedInstruction and the argument to which the operation is applied. The result is a Future-like data structure (CompletedStep), which represents the operation’s result. The actual result can be retrieved by a blocking call to CompletedStep’s ResultValue property. Listing 5.11 shows the type definitions involved.

```csharp
using DistrEx.Coordinator.Interface;

static partial class DistrEx.Coordinator.Coordinator {
    static CompletedStep<TResult> Do<TArgument, TResult>(TargetedInstruction<TArgument, TResult> targetedInstruction, TArgument argument) {
        // ...
    }
    // ...
}

class DistrEx.Coordinator.Interface.CompletedStep<TResult> {
    TResult ResultValue {
        get { /* ... */};
        // ...
    }
    // ...
}
```

Listing 5.11: Invoking DistrEx operations with arguments.

In the following, we will revisit the example sentences we formulated in Sec. 5.2 using DistrEx’s grammar G, and demonstrate how these are translated to C# code that uses the DistrEx API. For brevity, we will provide some code statements only once, and later assume these are part of the example programs presented in subsequent code listings.
Basic Statements

Previous example sentences referred to two execution targets $X_p, X_q$. In our example code we will be using $x_p$ and $x_q$ as the C# identifiers for $X_p$ and $X_q$, respectively. Similarly, previously elementary operations and elementary two-ply operations were frequently named $F_i, F_{i+1}, F_{i+2}, F_k, F_l$ or $T_i$. In example code we will be using the identifiers $f_0, f_1, f_2, f_k, f_l$ and $t_0$ for these respective operations. Since the code examples below do not focus on operations’ method bodies, we will only declare the necessary identifiers. Examples of assigning an operation to a variable are provided below, using the identifiers $\text{elementaryOperation}$ and $\text{elementaryTwoPlyOperation}$. Furthermore, we will assume the outermost composite operation to be named $\text{operation}$. Its argument and result types are $TArgument$ and $TResult$. Invoking this (composite) operation is done the Coordinator’s $\text{Do}$ method described above. Listing 5.12 shows these declarations and statements.

```csharp
using DistrEx.Common;
using DistrEx.Coordinator;
using DistrEx.Coordinator.Interface;

// Execution targets:
TargetSpec xp;
TargetSpec xq;

// Elementary operations
Instruction<TArgument, TResult> elementaryOperation =
    ( CancellationToken cancellationToken, Action reportProgress, TArgument argument ) => {
        // ...
    };
TwoPartInstruction<TArgument, TResult> elementaryTwoPlyOperation =
    ( CancellationToken cancellationToken, Action reportProgress, Action reportCompletedPart1, TArgument argument ) => {
        // ...
        reportCompletedPart1();
    };

// (composite) DistrEx operation:
TargetedInstruction<TArgument, TResult> operation;

// argument and result of (composite) DistrEx operation:
TArgument argument;
TResult result;

// Invoking a (composite) DistrEx operation
```
result = Coordinator.Do(operation, argument).ResultValue;

Listing 5.12: Invoking DistrEx operations with arguments.

Execution Targeting

Listing 5.13 shows the DistrEx equivalent to the sentence $X_p \cdot F_i$. Listing 5.14 shows the DistrEx equivalent to the sentences $X_p \cdot T_i$ and $X_p : T_i$.

Instruction<TArgument, TResult> f0;
operation = xp.Do(f0);

Listing 5.13: DistrEx equivalent of $X_p \cdot F_i$.

TwoPartInstruction<TArgument, TResult> t0;
operation = xp.Do(t0);
operation = xp.GetAsyncResult<TResult>();

Listing 5.14: DistrEx equivalent of $X_p \cdot T_i$ and $X_p : T_i$.

Sequential Composition

Listing 5.15 shows the DistrEx equivalent to the sentence $X_p \cdot F_i \rightarrow X_p \cdot F_{i+1} \rightarrow X_p \cdot F_{i+2}$.
Listing 5.16 shows the DistrEx equivalent to the sentence $X_p \cdot T_i \rightarrow X_q \cdot F_{i+1} \rightarrow X_p \cdot F_{i+2} \rightarrow X_p : T_i$.

Instruction<TArgument, TResult0> f0;
Instruction<TResult0, TResult1> f1;
Instruction<TResult1, TResult> f2;
operation = Coordinator.Do(xp.Do(f0)).ThenDo(xp.Do(f1)).ThenDo(xp.Do(f2));

Listing 5.15: DistrEx equivalent of $X_p \cdot F_i \rightarrow X_p \cdot F_{i+1} \rightarrow X_p \cdot F_{i+2}$.

TwoPartInstruction<TArgument, TResult> t0;
Instruction<Guid, TResult1> f1;
Instruction<TResult1, TResult> f2;
operation = Coordinator.Do(xp.Do(t0)).ThenDo(xq.Do(f1)).ThenDo(xp.Do(f2)).ThenDo(xp.GetAsyncResult());

Listing 5.16: DistrEx equivalent of $X_p \cdot T_i \rightarrow X_q \cdot F_{i+1} \rightarrow X_p \cdot F_{i+2} \rightarrow X_p : T_i$.
5. DistriEx

Parallel Composition

Listing 5.17 shows the DistriEx-equivalent to the sentence $X_p \cdot F_i \rightarrow X_p \cdot F_k \parallel X_q \cdot F_i \rightarrow X_q \cdot F_{i+1}$.

Instruction<TArgument, TResult0> f0;
Instruction<TResult0, TResultK> fk;
Instruction<TResult0, TResultL> fl;
Instruction<Tuple<TResultK, TResultL>, TResult> f1;
operation = Coordinator.Do(xp.Do(f0)).ThenDo(xp.Do(fk), xq.Do(fl)).ThenDo(xq.Do(f1));

Listing 5.17: DistriEx-equivalent of $X_p \cdot F_i \rightarrow X_p \cdot F_k \parallel X_q \cdot F_i \rightarrow X_q \cdot F_{i+1}$.

5.4 Evaluation

To date, DistriEx has dedicated limited resources to the adoption of DistriEx in the development process of NG. As a result, no extensive empirical analysis of DistriEx’s utility to the business case at hand can be made. So far, only one partially implemented distributed acceptance test case exists. From personal correspondence with this test case’s implementers, we have heard of a number of challenges that arise from using the current implementation of DistriEx:

• Poor performance when using large quantities of dependency assemblies.
• Attaching a debugger to a remote worker process is currently not integrated into DistriEx.
• Lack of support for non-statically defined operation delegates causes less concise programs.

Although DistriEx currently supports skipping the deployment of selected assemblies to increase execution performance, it is the creation of AppDomains for each operation execution that is computationally expensive. In a rapid edit compile execute cycle, this can limit the user’s productivity by slowing down the execution phase. Another challenge users face when developing DistriEx programs, is the lack of accessible support for debugger attachment. A debugger — a program for observing and modifying the execution of another running process — cannot be easily attached to operations being executed on a remote worker. The most accessible way of obtaining information from an executing operation’s internals currently is writing data to the worker process’s standard output. Finally, the lack of support for non-statically defined operation delegates primarily eliminates the users’ freedom to create operation delegates as compositions of delegates and variables. Instead, an operation’s entire configuration can only depend on its arguments. Arguably, this forces users to develop highly isolated units of behavior, but in practice users seem to value the flexibility that follows from using non-statically defined delegates.
Chapter 6

Conclusions and Future Work

In this chapter, Sec. 6.1 will first address the research questions we initially posed in Sec. 2.3.1, after which Section 6.2 will explain related areas that are open to future research. We conclude this chapter by reflecting upon our own work in Sec. 6.3.

6.1 Conclusions

6.1.1 Answering Research Question 1

In Ch. 2 and Ch. 3, the reader has been familiarized with the characteristics of the business case at hand, as well as the broader concepts of software test automation, and acceptance testing. While Ch. 3 discusses traits that a software engineering effort should ideally exhibit, Ch. 2 regarding the business case reveals a state of affairs that does not match this ideal. In Sec. 2.3 we established that currently the only accessible way to improve the state of affairs in the area of testing is through acceptance testing. However, Sec. 3.2.2 stresses the importance of appropriate testing at different levels, i.e. the importance of integration testing for a fruitful acceptance testing effort. Seemingly, this raises a contradiction, which we challenge by posing R.Q. 1:

Research Question 1 Does acceptance testing retain its utility if integration tests are lacking?

To our knowledge, this idea is not considered in literature, hence we must rely on our own observations from which to derive an answer. From the business case at hand, we have learned that both manual and automated acceptance tests have revealed regression bugs on multiple occasions. Specifically, automated acceptance tests have revealed the SUT’s failure to perform essential tasks on numerous occasions, while unit tests had not revealed these faults. Given the status-quo in unit testing within this business case, as it is presented in Sec. 2.2.1, a more extensive unit testing suite might have revealed such faults in the SUT, rendering acceptance tests silly for this purpose. However, given the substantial overlap in method coverage between unit tests and acceptance tests, and the fact that faults were indeed revealed using acceptance tests, we propose to answer R.Q. 1 affirmatively.

53
6. Conclusions and Future Work

We derive this conclusion as follows: Firstly, had there not been such an extensive overlap\(^1\) in covered methods between the discussed forms of testing, the additional coverage achieved through acceptance testing would at least have installed more confidence in the SUT as a whole, if no faults would have been revealed by acceptance testing, as opposed to not having any certainty about the otherwise untested parts of the SUT’s behavior. Secondly, even in areas where unit testing and acceptance testing overlap, our anecdotal evidence shows that it proved useful in revealing faults that by unit testing alone were not discovered. Both arguments demonstrate the utility of acceptance testing in the face of lacking integration tests.

6.1.2 Answering Research Question 2

Research Question 2 Can meaningful acceptance tests for distributed systems be formulated?

Restating the strategy for answering R.Q. 2 from Sec. 2.3.1, we argue that this question will be answered if R.Q. 2.1, R.Q. 2.2 and R.Q. 2.3 have been answered. These questions will be answered in the following subsections.

6.1.3 Answering Research Question 2.1

Research Question 2.1 How can a test program interact with a distributed system?

Chapter 4 compares testing a distributed system to acceptance testing. By executing the SUT separately from the test program, the SUT will be able to run in an unmodified configuration during test. In order for a test program to interact with the distributed SUT, Ch. 4 explains the need for a separate distributed test system that can execute such interactions across multiple machines. Additionally, in order to properly test concurrent behavior, Sec. 4.4 introduces the concept of concurrent composition, a semantic construct for expressing the concurrent execution of operations, amidst otherwise sequentially arranged operations.

We consider this extended model for specifying behavior, comprising sequential and concurrent composition of operations, as well as operation targeting to be an answer to R.Q. 2.1. By utilizing these concepts in test programs, we possess sufficient expressive power formulate test programs for testing distributed systems.

6.1.4 Answering Research Question 2.2

Research Question 2.2 Do such interactions leave test validity intact?

As stated in Sec. 6.1.3, the presented approach to testing treats the SUT as a black box that is interacted with using intended channels only. By ‘intended channels’ we mean ways

\(^1\)Here, we consider coverage granularity at the level of methods, since we assume that on average, every covered method is at least executed by its most common path, which is likely to be the most relevant path.
of interacting with the SUT that were explicitly foreseen and allowed by its creator. By definition, we argue, such interactions are within the perimeter of the SUT’s functional specification, and can thereby only cause the SUT to react to interactions within said perimeter. As such, there cannot be any unintended alterations to the SUT’s behavior resulting from interactions that are explicitly allowed within this perimeter.

In this reasoning, the concept of a functional specification is key. As Ch. 3 argues, SUT can only be sensibly tested with respect to how it conforms to its functional specification. When designing tests, it is the tester’s duty to be fully aware of the consequences that each interaction with the SUT has, according to its specification. Without this awareness the tester cannot design tests that truly verify the SUT’s conformance to its specification. In summary, we answer R.Q. 2.2 positively: Interactions of the kind discussed above do not impact the SUT’s behavior in any way that jeopardizes test validity.

6.1.5 Answering Research Question 2.3

Research Question 2.3 Can a notation for writing distributed acceptance tests be developed?

In Sec. 5.2 we have proposed a notation that lets us express distributed and concurrent behavior by using three binary operators to communicate three corresponding concepts: • for execution targeting, → for sequential composition, and ∥ for parallel composition. Section 5.3 proposes the implementation of these operators as API functions that can be used to engineer distributed programs. We therefor conclude that both a formal notation, and a working implementation thereof can, and has been developed, which answers R.Q. 2.3 positively.

6.2 Future Work

Some aspects of this work deserve further attention. We classify the opportunities for future work by two categories: Theoretical and practical work. Section 6.2.1 and Sec. 6.2.2 explore each category separately.

6.2.1 Theoretical Work

In its current state, the production rules of the formal grammar $G$, which is the foundation of the DistrEx implementation, do not all guarantee the creation of semantically sound sentences. Specifically, the execution targeting operator $\vdash$ defined in P.R. 6 can be used inappropriately. Currently, there is no syntactical construct that guarantees that both execution phases of a two-ply operation $T_i$ are targeted at the same execution target. By redesigning the syntax of $G$, the need for semantic checking could become obsolete, making the implementation of $G$ as a software language easier.

Furthermore, $G$ currently only supports composing operations sequentially and parallelly. Its expressive power may be extended by including operators for control structures,

\footnote{Testing a system’s resilience to disallowed interactions is a different subject that is left out for brevity.}
such as decisions (‘if then else’) and iterations (‘for’ and ‘while’ loops). The integration of control structure operators into \( G \) will let users specify full-fledged distributed programs that do not rely on external means for control flow composition. For the user, this may result in more concise programs, and more efficient program execution.

On a higher level, an analysis on the (perceived) utility of the DistrEx notation, and how user friendly it is considered to be can create valuable insight into its design quality. Such an investigation might yield an objective account of the non-functional requirement of ease of use we briefly mentioned in Sec. 5.1.

6.2.2 Practical Work

Regardless of the : operator’s syntactic weakness, an implementation should reflect its formal foundation accurately. Currently the : operator is implemented by the GetAsyncResult method and relies on the user to supply the appropriate return type parameter. This parameter should instead be derived, either by the C# compiler, or by the DistrEx runtime system, since this information is contained in a composite operation’s structure.

Secondly, we expect significantly more concise DistrEx programs from operation composition that is denoted using binary operators. In C#, a number of existing binary operators can be overloaded to operate on types that are not native to the C# language. By applying this technique to DistrEx’s TargetedInstruction, operator composition can be expressed through binary operations that resemble DistrEx’s intrinsic grammar \( G \) much closer than its current fluent API. A second means for enabling users to write more concise DistrEx programs, is by adding support for non-statically defined operation delegates.

Furthermore, the application domain of DistrEx may be extended to cater for reactive applications by building the reactive programming model into it. Currently, DistrEx operations are functions that are applied once to transform a single argument into a single result value. The reactive programming model allows functions to be applied to streams of argument values, and as a consequence produce streams of result values. By combining this model with DistrEx’s execution targeting paradigm, a more powerful programming model might emerge.

Lastly, the current implementation of DistrEx does not consider any non-functional requirements, such as security or performance. Since these were of no critical concern to the business case at hand, these subjects have received no attention yet. In order to make DistrEx more widely applicable, efforts in developing these aspects might be required.

6.3 Threats to Validity

Initially, we provide a brief overview of the software engineering characteristics of the business case at hand. However, the presented facts are not put into perspective by relating them to other (similar) software development projects’ characteristics. Consequently, the role of these facts in objectively supporting the sentiment that contributors are too unfamiliar with the codebase may be disputed.

Secondly, this work regards acceptance testing as the most accessible means to improving an application’s test suite. Based, in part, on anecdotal evidence, it is undoubtedly one
accessible means for achieving this goal, but likely not the only one. This work lacks an in-depth analysis of applicable strategies to improving the test suite of the business case at hand. Such an analysis could provide insight into a variety ways to achieve this goal, accompanied by estimates regarding each particular strategy’s required effort and success.

Furthermore, the presented model for interacting with distributed systems draws its inspiration from acceptance testing, where the SUT is treated like a black box. This work does not explore any alternative models, nor does it motivate this lack of alternatives.

Finally, the developed notation and subsequent implementation thereof are likely to depend on preconceptions originating from the business case at hand. Throughout, no attempt at identifying such preconceptions is made, from which it follows that results of this work are likely to have a limited applicability.

Ultimately, the decisions that have led to the content and structure of this work were implicitly driven by an interest in engineering activities. In particular, developing a technical solution to the testing challenges of the business case at hand has captured the authors’ attention. As a result from following this path, a number of subjects have deliberately been left open for further discussion.
Chapter 7

Related Work

7.1 Automated Testing

In software testing, automated testing is a form of testing that tests a (part of) a software system entirely autonomously, which often manifests itself as a form of unit testing. Originally defined in [31], the unit testing standard expects that unit test execution be automated and therefore repeatable. This approach has been successfully adopted by Beck [7] resulting in the development of SUnit, a Smalltalk unit testing framework. Inspired by SUnit, numerous similar unit testing frameworks have been developed for other languages, most relevant of which are perhaps JUnit [9] for Java programs and NUnit [55] for .NET programs. Frameworks descending from SUnit use a common set of formalisms to facilitate automated test execution. Key elements are a test runner program that discovers tests, executes them, and reports test execution results. Test execution results are produced by a mechanism to assert behavioral conformance or non-conformance, identified as a pass or failure, respectively.

7.1.1 Flavors of Automated Testing

Automated testing is not limited to testing small, cohesive units. Where unit testing strictly only spans the scope of small units, e.g. at most a class, the automated testing paradigm it is based on applies equally well to integration testing [60], and acceptance testing [43].

Motivated by the business case at hand, this work is primarily concerned with automated acceptance testing. Even though Park and Maurer [53] identify a wealth of sources advocating acceptance testing through a testing bus, the presented business case does not fit this particular model. Instead, our approach to acceptance testing involves prominent use of a GUI testing toolkit, which is known as GUI Testing. Literature on the subject distinguishes three flavors of GUI Testing: Record and Replay (R&R) testing, where test cases are generated by recording a user’s interactions with the system; Model-based test case generation [67, 46]; And manual test case generation [45, 18, 29]. In the light of the business case at hand, the additional dimension of temporal synchronization to GUI Testing introduced in [1] is of negligible relevance, given the Finite State Machine (FSM) inspired modeling nature [2] of acceptance tests.
We are not aware of any publications discussing acceptance testing of distributed applications.

7.2 Concurrent Computing

In the domain of concurrent computing, communication mechanisms fall into either one of two categories. The first, shared memory communication, allows processes to access and communicate through shared data structures, but it is inherently challenging to scale to large, distributed systems. Motivated by a notion stating that shared memory distributed systems are easier to program, Distributed Shared Memory (DSM) systems have been developed. DSM systems create an illusion of shared memory in distributed systems, often implemented by underlying message passing systems. Nitzberg and Lo [50] compactly discuss characteristics of different DSM systems, and the considerations that play a role in the design of such a system.

The second category, message passing communication, lets processes communicate only through sending and receiving messages along clearly defined paths of communication. A prime example of message passing-based concurrency is the Actor Model [36, 35]. Systems of either category can be simulated on top of the other, relieving system designers of any restrictions that a system’s target environment might have. Section 7.2.1 further explores this subject in the context of RPCs.

A different way of classifying concurrent systems is by problem decomposition, rather than underlying means of communication. Oftentimes, a problem or algorithm can be decomposed into smaller problems that are connected through operational operators. This is the philosophy of Process Algebra [11], a field studying strict rule based composition of (concurrent) processes. On the practical side of concurrent process composition there is the concept of Algorithmic Skeleton Frameworks (ASKFs) [25]. An ASKF is a programming model implemented as a set of higher order functions that can be used to compose a concurrent process.

DistrEx can be considered an ASKF, while its intrinsic grammar $G$ can be considered a Process Algebra. However, we are not aware of any work that proposes a Process Algebra with explicit execution targeting.

7.2.1 Writing Concurrent Programs

Perhaps the topic within the field of concurrent computing most relevant to this work, is that of writing concurrent programs. Where imperative (object-oriented) programming languages let programmers write units of sequential code suitably well, it is often much harder to write concurrent programs using such a language. Due to the sequential nature of program code, the intentions of a concurrent program are inherently less visible from the code alone than the intentions of a sequential program would be. Moreover, the readability and expressiveness of such code must often be sacrificed even more to accommodate additional boilerplate code required for concurrent programs. A minority of programming languages, patterns and modeling techniques truly support writing concurrent programs without the need to sacrifice expressiveness and clarity.
Early attempts at simplifying the design of concurrent programs involve the creation of libraries and programming languages with built-in support for concepts related to concurrent programming. The former includes ASKFs, on which González-Vélez and Leyton [32] provide a survey discussing contemporary work. Their publication contrasts ASKFs and common programming patterns used for implementing parallelism. Examples of the latter are the Argus [40], Emerald [16] and Eden [15] programming languages. Argus supports shared versioned objects, as well as atomic operations, while Emerald and particularly Eden relieve the programmer of facing distributed programming challenges, such as network failure and replication. Synchronized remote procedure calls [70] are commonly used to facilitate the distributed face of concurrent programming.

The Actor Model [35] eliminates many problems of shared-memory concurrent programming by requiring all concurrent lines of execution to communicate through message passing. Compared to traditional concurrent systems, writing concurrent actor systems is arguably more intuitive, since actors’ lines of execution are more clearly separated due to the lack of shared objects. At the same time, the lack of shared objects introduces an entirely different set of challenges, such as the lack of a global state. Commonly, actor systems allow for great flexibility in network topology through the means of sending actor references as messages.


However, programmers using this kind of technology can often merely offload the need for boilerplate code. The actual caveats of concurrent programming, such as deadlocks and starvation, still require the programmer’s careful attention. Benton et al. [10] argue that programmers writing concurrent programs need to be supported in that endeavor, beginning with the language they are writing in. Yet, many contemporary object-oriented languages support only a limited set of concurrency concepts, and therefore offer insufficient expressiveness with respect to concurrency.

Process Calculi (or Process Algebras) [11] on the other hand are particularly expressive constructs for specifying concurrent processes [6]. The axiomatic approach to process composition is also well-suited for adaptation into DSLs allowing for bridging the gap between the highly theoretical concept of any Process Calculus and its practical usage by programmers. This approach is taken in e.g. Polyphonic C# [10], Join Java [69] and Funnel [51], where domain specific concepts from the Join Calculus [30] are added to existing programming languages. Similarly, the former Business Process Modeling Language (BPML) [37] and Ateji PX [68] are examples of (programming) languages that can express concepts from the π-calculus [48].

DSLs can even appear as visual languages. Instead of using text to specify behavior, a visual notation, i.e., diagrams, is used to capture process semantics. An example of potentially suitable diagrams are Petri nets [54], which have been shown to be (partially) translatable to Join Calculus [4]. The Unified Modeling Language (UML) [52] also comprises visual notations suitable for expressing concurrency, as discussed in [61]. However, to our knowledge no visual notations for capturing concurrent semantics have been implemented as an
execuable language.

All the aforementioned techniques have in common that they are designed for concurrent processes with generic decomposition characteristics, i.e. the runtime system at hand decides how to break down larger problems into smaller ones. However, these referenced works’ relevance in that respect is not to be misinterpreted as the reason for their appearance in this section. Analogously, the referenced works’ relevance to the study of communicating processes is not the primary motivator for their occurrence in this subsection. Here, we are primarily concerned with the means a given technique offers programmers for specifying concurrent behavior, and the responsibilities it can relieve the programmer of in the process. Simultaneously, the business case at hand demands that the user explicitly specifies the execution target of operations, as opposed to the runtime system making this decision.

Particularly useful to batch processing are the Pipeline Programming Model [64] and the Split-Merge Model, to both of which this work’s execution diagrams show conceptual resemblance. This resemblance is intentional, since both models are easy to understand for seasoned programmers. Especially the Pipeline Programming Model requires little to no specialized knowledge from a programmer, as it is closely related to function composition, an essential concept of any programming language.

Remote Procedure Calls

Part of a coordinated distributed application is instructing one machine from another, i.e. requesting a machine to perform some kind of computation for the requester. This concept, named RPC, is a form of Inter-process Communication (IPC) [71, 14] that is conceptually best represented as a form of message passing IPC. All RPC implementations follow a pattern of binding stubs to (remote) interface implementations. RPC users on both ends of the line of communication only interact with stubs, while the communication occurs solely between said stubs [14, 63]. That way, all communication is transparently taken care of by the RPC implementation.

Literature distinguishes two kinds of RPC: blocking and non-blocking method calls [63], but we argue that this distinction is not specific to RPC. In this work we are primarily concerned with asynchronous, i.e. non-blocking RPC.

Dynamically Changing Behavior

One limitation of RPC is that an exported, i.e. remotely accessible, interface exclusively exposes a predefined set of interaction points. Such interaction points are methods that can be invoked on the interface implementation. In order to achieve greater flexibility in solving the business case at hand, we investigate options of lifting the limit on interaction points to accommodate an arbitrary number of interaction points that are defined at runtime.

One way to achieve this is through runtime program modification [27, 57], replacing the exported interface’s implementation with a different one before each RPC execution. Practically, a contemporary approach to runtime program modification is through Reflection [28].

62


Appendix A

NG Codebase Metrics

In order to better understand the magnitude and complexity of the NG application, we have applied repository mining techniques to the NG source code repository. Through this process we revealed the size of, and contributors’ involvement in, the code base. Through visually comprehensive presentations of extracted data, the reader can intuitively understand the information contained within the source code repository. Additionally, we have investigated test coverage reports of the unit test suite and acceptance test suite. Techniques used for extracting and representing relevant data from these reports are similar to the employed repository mining techniques. This chapter provides background information on the process of gathering and visualizing the previously presented data.

The codebase’s complexity will be discussed in Sec. A.1, followed by the topics of test coverage in Sec. A.2, and contributor’s involvement in Sec. A.3. Section A.4 addresses miscellaneous concerns the reader might have regarding the content of this chapter.

A.1 Complexity

In the context of this section, we consider complexity a matter of size. We established the plain line count directly from the source code files using the word count (wc) command with the line count option. In this process, we selected only source code (text) files with a file name extension matching \texttt{cfg}, \texttt{config}, \texttt{controlsettings}, \texttt{cpp}, \texttt{cs}, \texttt{json}, \texttt{rb}, \texttt{resx}, \texttt{settings}, \texttt{txt}, \texttt{xaml} or \texttt{xml}. Those metrics that require semantic awareness of \texttt{C#} code, such as class count, method count, or even coverable line count, were collected from information contained in OpenCover coverage reports. The use of these coverage reports is explored further in Sec. A.2.1. The data gathered in this process is presented in Sec. 2.2.2.

A.2 Test Coverage

In this section we will discuss the difference in test coverage achieved by unit testing and acceptance testing. The test coverage information we have collected was generated by OpenCover. OpenCover produces line and branch coverage of which we will only consider the former.
A. NG Codebase Metrics

A.2.1 Data Aggregation and Processing

OpenCover produces Extensible Markup Language (XML) files containing the coverage data that was collected during a test run. The information contained within these report files reveals which assemblies are required to execute the test run, which types within each assembly are used and which methods are contained within a type. At each level in this hierarchy, information about the line coverage is collected.

We executed two test runs, one for unit tests and one for acceptance tests, and integrated the results into one large data set. From this collective data set, we compiled a list of assemblies that were identified by OpenCover. For each assembly, we then computed the total overlap in coverage between unit tests and acceptance tests. I.e. for each line, we discovered whether it had been covered during a unit test run only, during an acceptance test run only, covered by both, or not covered at all. We do not make use of available information on how often each line was covered during each test run. The resulting data set is presented in Sec. 2.2.1 in part by Fig. 2.2.

A.3 Contributors’ Involvement

The goal of this section is to reveal which contributors have contributed to which parts of the codebase, and how large contributions were. Since the source code repository is maintained using Git, this section will use Git terminology to refer to concepts such as contributions (commits), contributors (authors, committers), and parts of the codebase (files).

A.3.1 Data Aggregation and Processing

For the purpose of this investigation we have listed the set of all 8,533 commits in the period from March 4th, 2012 to March 4th, 2013. From this set, we have extracted all 29,733 modified files, as well as all 51 committers. Additionally we have counted the accumulative length of all users’ individual contributions to each file, within this set of commits. This reveals how involved each user was with the development of each file during the given period. This measure of involvement is expressed as the number of changed lines for a file. It is worth mentioning that in our counting process, we attributed contributions to their committer, rather than their author. We justify this choice by observing that we included development commits, as well as merge commits, and the fact that management dictates that merge commits require their committer to review coworkers’ contributions. We believe that given this reviewing process, our count appropriately attributes contributions to both their authors and reviewers.

To this data set we subsequently applied filters, eliminating all contributions to files that involve zero changed lines (empty file additions), and all contributions to files that are not source code files. Files with a file name extension matching .cfg, .config, .controlsettings, .cpp, .cs, .json, .rb, .resx, .settings, .txt, .xaml or .xml are considered source code files. After filtering, 25,375 different files and 51 authors remained in the data set.
A.3.2 Data Visualization

The data collected by the aforementioned process — containing information for each file — is very fine-grained. It is therefore sensible and impractical to communicate this information to the reader in full detail. Hence, we seek to reduce the granularity from the level of files to the level of assemblies. Ultimately, our goal is to produce a matrix from the filtered data set displaying for each assembly how involved each contributor was in its evolution. Moreover, we want to differentiate each contributor’s involvement with test code and production code. The resulting plot is presented in Fig. 2.3.

Before we can conceive a fitting representation for the matrix’s contents, we need to gain insight into the distribution of differently sized contributions. To this end, we plot the accumulative contribution size for each assembly, for each contributor, including the group of contributions belonging to no particular assembly. Here, we maintain the distinction between test- and non-test code files.

Figure A.1 displays these plots: Each discrete unit along the horizontal axis represents the accumulative contribution that one particular contributor made to one particular assembly. The vertical axis indicates the accumulative contribution’s size. Note that the vertical axes use a logarithmic scale.

Since contributions are sorted by size, contributions of equal size are plotted adjacently. Therefore, frequently occurring sizes are identified by wide horizontal sections of the same height. E.g. size 2 occurs 93 times in Fig. A.1(a) and 48 times in Fig. A.1(b).

By inspecting Fig. A.1 we observe that all plotted contribution sizes are smoothly distributed over a logarithmic scale. It is for this reason that, we use colors along gradients to represent contribution sizes’ logarithms. Again, we will distinguish application code and test code, for each of which we have chosen a separate color scheme. Figure 2.4 shows the color scheme we have chosen to visualize contribution sizes for application code and test code, and the final matrix of data points is presented in Fig. 2.3.

A.4 Miscellaneous Remarks

Regarding the distinction between Intersite’s internally developed dependency assemblies made in Sec. 2.2.2, we report that this is based on information collected during an informal survey among contributors. Results from this survey were manually evaluated, leading to the classification of dependency assemblies.

Regarding the analysis of the source code repository, we report that by its pure analysis, we were only able to collect information about contributors and files. This information was subsequently augmented by adding to it the assembly that each file belongs to. For this purpose, a list of assemblies and their location on the file system has been compiled manually, after which files and assemblies were joined on matching paths.
Figure A.1: Contribution sizes of the NG repository. A.1(a) shows contributions to application code. A.1(b) shows contributions to test code. Each column in each of the plots represents the accumulative contribution of a particular user to a particular assembly. The height of each column represents contribution size.
Appendix B

Glossary

Symbols

\( \lambda \)-calculus

A formal system to express computation based on function abstraction and application. \[28\]

.NET Framework

A software framework by Microsoft. \[81\]

A

Actor Model

A mathematical model of concurrent computation based on message-passing interaction. \[60, 61, 74, 75\]

Algorithmic Skeleton Framework

An implementation of a high-level programming model. \[81\]

AppDomain

An isolated section of a .NET application’s memory, into which assemblies are loaded and memory allocations are performed. \[47, 49, 52\]

Application Programming Interface

A specification describing how a software component can be interacted with by other software components. \[81\]

Argus

A CLU-based programming language supporting the creation of concurrent programs. \[61\]
GLOSSARY

Ateji PX
Extension of the Java programming language with support for π-calculus concepts. 61

B
Backus-Naur Form
A notation form for specifying production rules of context-free grammars. 81

C
C++
An object-oriented programming language. Pronunciation: “C plus plus”. 74

C++/
A programming pattern for the C++ programming language implementing the Actor Model. Pronunciation: “C++ parallel”. 61

C#
An object-oriented programming language. Pronunciation: “C sharp”. 7 8 23 27 28 36 38 41 44 46 49 50 56 69 77

CLU
A programming language containing a few object-oriented concepts. 73

D
DistrEx
The central subject of this work: A framework for composing distributed programs. 3 37 49 51 52 55 56 60

Domain Specific Language
A software language tailored to express concepts related to a specific application domain. 81

Dynamic Link Library
A binary file containing compiled code. 82

E
Eden
A Haskell-based programming language supporting the creation of concurrent programs. 61
Eiffel
An object-oriented programming language. [75]

Eiffel/
A programming pattern for the Eiffel programming language implementing the Actor Model. Pronunciation: “Eiffel parallel”. [61]

Emerald
An object-oriented programming language supporting the creation of concurrent programs. [61]

Extensible Markup Language
A markup language for creating documents in a machine- and human readable format. [84]

F

Finite State Machine
A State Machine with a finite number of states. [82]

Fugro Intersite B.V.
A subsidiary of Fugro N.V. (Fugro), registered with number 34048179 in the company register of the Dutch chamber of commerce. [82]

Fugro N.V.
A multinational corporation specialized in providing geotechnical and survey services. Fugro N.V. is publicly traded at Euronext under the name FUR. [82]

Functional Nets
A functional programming paradigm based on Petri nets and the Join Calculus. [75]

Funnel
A programming language based on Functional Nets. [61]

G

Git
A revision control and source code management tool. [11, 70]

Global Navigation Satellite System
A system of satellites that allow a receiver unit to calculate its geo-spatial position. [82]
GLOSSARY

Graphical User Interface
A type of user interface that lets the user interact with a system through visually/semantically presented components. 82

GUI Testing
A testing method that simulates a user interacting with the system through its GUI. 59, 78

H

Happened-Before relation
A relation defining the temporal ordering of events. 82

Haskell
A functional programming language. 74

I

Inter-process Communication
Communication between processes. 82

Introspection
The ability to inspect a program’s structure and behavior. 78

J

Java
An object-oriented programming language. 74, 76

Join Calculus
A Process Calculus. 61, 75

Join Java
An extension of the Java programming language containing concurrency abstractions. 61

JUnit
A unit testing framework for the Java programming language. 59

M

M.V. Fugro Galaxy
A surveying vessel owned by Fugro Galaxy Inc., registered in Nassau, Commonwealth of the Bahamas. 82
Message Manager
   A proprietary message-passing middleware by [Intersite] which employs [Pragmatic General Multicast (PGM)] [83]

Microsoft
   A multinational corporation that predominantly produces software products. [7] [15] [45] [79]

N
Network Time Protocol
   A protocol for synchronizing clocks of network-connected computers. [83]

NUnit
   A unit testing framework for [NET] programs. [59]

O
Open Systems Interconnection Model
   A standardized (ISO/IEC 7498-1) conceptual partitioning of communication systems into layers. [83]

OpenCover
   A code coverage tool for code targeting [NET] 2.0 and newer versions. [7] [8] [69] [70]

Oxford English Dictionary
   A dictionary of the English language, published by the Oxford University Press. [83]

P
Petri net
   A model for concurrent processes. [61] [75]

π-calculus
   A Process Calculus. [61] [74]

Pipeline Programming Model
   A model for composing multi-stage processes. Concurrency can be achieved by processing multiple work units in concurrently operating stages. [62]

Polyphonic C#
   An extension of the [C#] programming language containing concurrency abstractions. [61]
**Pragmatic General Multicast**

Often referred to as reliable multicast, a computer network protocol. 83

**Process Algebra**

Process Calculus 60 61

**Process Calculus**

Algebra capable of expressing concurrent processes. 61 76 77

**R**

**Real-Time**

A constraint on computing systems stating that operations must be completed before a certain deadline. 6

**Record and Replay**

A form of GUI Testing. 83

**Reflection**

An expansion of Introspection that adds the ability to modify a program’s structure and behavior. 45 47 62

**Remote Procedure Call**

A form of IPC where one process instructs another process to perform some computation for it. 83

**RS-232**

A family of standards for serial binary communication between electronic devices. 5

**S**

**Server Message Block**

A protocol for sharing files and devices over a network. 84

**Service-Oriented Architecture**

An application programming model suitable for expressing decoupled, modular and network-connected systems. 84

**Smalltalk**

An object-oriented programming language. 59 79

**Split-Merge Model**

A model for decomposing a work unit into smaller work units that are processed concurrently. 62
SQLite
A file-based relational database management system. [7]

Starfix.NG
Software system for offshore surveying, developed by Intersite. [5] [83]

State Machine
A model of computation involving states and transitions between states. [75]

SUnit
A unit testing framework for the Smalltalk programming language. [59]

T
Transport Layer
Level 4 of the Open Systems Interconnection (OSI) Model. [5] [79]

U
Universal Serial Bus
A standard specifying the connection between computers and peripheral devices. [84]

User Datagram Protocol
A Transport Layer protocol. [84]

W
Windows 7
An operating system by Microsoft. [7]

Windows Automation API
A library by Microsoft that eases simulating user interactions with Windows applications. [15]

Windows Communication Foundation
A collection of APIs for building Service-Oriented Architecture (SOA) applications. [84]

X
x86-64
A 64 bit microprocessor architecture. [7]
Appendix C

Acronyms

H I J N O P S U X Symbols L M

Symbols

.NET

The .NET Framework 7, 27, 47, 59, 73, 77

A

API

Application Programming Interface 36, 38, 41, 45, 46, 48, 49, 53, 56, 79

ASKF

Algorithmic Skeleton Framework 60, 61

B

BNF

Backus-Naur Form 28

BPML

Business Process Modeling Language 61

C

COTS

Commercial Off-The-Shelf 7, 12

D

DSL

Domain Specific Language 37, 61
ACRONYMS

**DSM**
- Distributed Shared Memory [60]

**Dynamic Link Library**
- Dynamic Link Library [46]

**F**

**FSM**
- Finite State Machine [59]

**Fugro**
- Fugro N.V. [75]

**G**

**Galaxy**
- M.V. Fugro Galaxy [5]

**GNSS**

**GUI**
- Graphical User Interface [15, 21, 59, 76]

**H**

**HB relation**
- Happened-Before relation [24, 29, 31]

**I**

**Intersite**
- Fugro Intersite B.V. [2, 5, 7, 12, 14, 52, 71, 76, 78]

**IPC**
- Inter-process Communication [62, 78]

**K**

**KLOC**
- Kilo LOC [7]

**L**
- 82
LAN
Local Area Network [7]

LOC
Line(s) of Code [8, 82]

M

MM
Message Manager [7]

N

NG
Starfix.NG [2, 5, 9, 11, 12, 14, 17, 25, 33, 36, 52, 69, 72]

NTP
Network Time Protocol [7]

O

OED
Oxford English Dictionary [23]

OSI Model
Open Systems Interconnection Model [79]

OSS
OSS [7, 12]

P

PGM
Pragmatic General Multicast [76]

R

RPC
Remote Procedure Call [26, 60, 62]

R&R
Record and Replay [59]

S
ACRONYMS

SMB
Server Message Block 7

SOA
Service-Oriented Architecture 79

SUT
System Under Test 15 17 19 22 25 32 33 44 53 55 57

T
TDD
Test Driven Development 19

U
UDP
User Datagram Protocol 5

UI
User Interface 20 21

UML
Unified Modeling Language 61

USB
Universal Serial Bus 5

W
WCF
Windows Communication Foundation 45

X
XML
Extensible Markup Language 70