A Language-Recognition Approach to Unit Testing Message-Passing Systems

IFEANYI W. UBAH
A Language-recognition Approach to Unit Testing Message-Passing Systems

Ifeanyi W. Ubah

Master of Science Thesis in Information and Communication Technology
Supervisor: Lars Kroll
Examiner: Jim Dowling
Contents

1 Introduction 1
  1.1 Problem Statement .................................. 3
  1.2 Contributions ...................................... 3
  1.3 Goals ............................................... 3
  1.4 Scope ............................................... 4
  1.5 Methodology ........................................ 4
  1.6 Outline ............................................. 4

2 Background 5
  2.1 Automated Software Testing .......................... 5
  2.2 Executions and the I/O Model ........................ 6
    2.2.1 Events ......................................... 6
    2.2.2 Executions ..................................... 6
    2.2.3 Behaviors ...................................... 7
  2.3 Formal Languages ................................... 7
    2.3.1 Formal Languages ................................ 7
    2.3.2 Grammars ....................................... 8
    2.3.3 Regular Grammars ................................. 9
    2.3.4 Regular Expressions ............................... 9
    2.3.5 Context-free Grammars ......................... 10
  2.4 Finite Automata .................................... 10
    2.4.1 Deterministic Finite Automata .................. 10
    2.4.2 Nondeterministic Finite Automata ............... 11
    2.4.3 NFAs with Epsilon-Transitions .................. 12
    2.4.4 Concatenating Finite Automata .................. 13
  2.5 The Kompics Component Model ....................... 14
    2.5.1 Ports ............................................ 14
    2.5.2 Channels ....................................... 15
    2.5.3 Event Handlers .................................. 16
    2.5.4 Components ..................................... 17
  2.6 Related Work ........................................ 17
### A Framework for Testing Message-Passing Systems

#### 3.1 The Language of Event Streams

#### 3.2 A DSL for Writing Specifications

#### 3.3 Regular Tests

- 3.3.1 Concatenating Executions
- 3.3.2 Union of Executions

#### 3.4 Extending Regular Tests

- 3.4.1 Blocks
- 3.4.2 Repeating Executions
- 3.4.3 The Kleene Closure on Executions
- 3.4.4 Unordered Executions
- 3.4.5 Specifying Constraints and Requirements on Executions

#### 3.5 Dependencies

- 3.5.1 Testing with External Dependencies
- 3.5.2 Component Mocking
- 3.5.3 Behavioral Mocking

#### 3.6 White Box Testing

- 3.6.1 Scheduling Inspections

#### 3.7 Matching Events

#### 3.8 Beyond Regular Tests

#### 3.9 Ambiguous Test Specifications

- 3.9.1 Using Matching Functions

#### 3.10 Terminating Test Cases

### KompicsTesting - An Implementation

#### 4.1 Ports and Directions

#### 4.2 Runtime

- 4.2.1 The Proxy Component
- 4.2.2 Internal Port Implementation in Kompics
- 4.2.3 Intercepting Messages
- 4.2.4 Scheduling

#### 4.3 Specification Builder

#### 4.4 Creating and Executing a Test Case

#### 4.5 Answering Requests

#### 4.6 Applications

- 4.6.1 The Kompics FSM
- 4.6.2 A Streaming Application

### Conclusions and Future Work

#### 5.1 Future Work

- 5.1.1 Implementation improvements

#### 5.2 Benefits, Ethics and Sustainability
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A DFA recognizing the sequence “cat”.</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>A NFA describing the language {ab, ac}.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>An ε-NFA describing the language {ab, ac}.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>A DFA recognizing the sequence “ac”.</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>A DFA recognizing the sequence “dc”.</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Concatenating automata in figures 2.4 with 2.5 recognizing the sequence “acdc”.</td>
<td>13</td>
</tr>
<tr>
<td>2.7</td>
<td>Port <code>pingpong</code> allows <code>ping</code> and <code>pong</code> events in the negative and positive directions respectively.</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>A providing component (ponger) has a negative inside port and positive outside port while a requiring component (pinger) has a positive inside port and negative outside port. Channels connect ports of opposite directions, (here + and −).</td>
<td>15</td>
</tr>
<tr>
<td>2.9</td>
<td>Event handlers are depicted with rounded rectangles. Here handler <code>handlePong</code> is subscribed to port <code>pingpong</code>. Outgoing events are triggered on inside ports and depicted with dash-arrows and diamonds.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>An automaton recognizing the same language as (e_1e_2e_3e_4^*).</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>An automaton recognizing a single event (e_i).</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Automaton for recognizing the concatenation of events.</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>NFA for conditional statement: “either expect (e_1) or expect (e_1) end”.</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>ε-NFA for “repeat body expect (e_1) end”.</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>An automaton for “unordered (e_1) end”. key: {(e_1, e_2)}.</td>
<td>29</td>
</tr>
<tr>
<td>3.7</td>
<td>EFSM for unordered executions.</td>
<td>30</td>
</tr>
<tr>
<td>3.8</td>
<td>Automaton for “repeat i body expect (e_1) end”.</td>
<td>31</td>
</tr>
<tr>
<td>3.9</td>
<td>Extending Figure 3.8 with headers “allow (e_1) drop (e_5)”.</td>
<td>32</td>
</tr>
<tr>
<td>3.10</td>
<td>DFA for “repeat i blockExpect (e_1) body expect (e_1) end”; key {next deterministic event, bitstring}.</td>
<td>33</td>
</tr>
<tr>
<td>3.11</td>
<td>EFSM for Figure 3.10.</td>
<td>34</td>
</tr>
<tr>
<td>3.12</td>
<td>An automaton incorporating an active state (q_1) that sends messages (m_2) and (m_3) upon entry and then transitions to next state (q_2).</td>
<td>35</td>
</tr>
<tr>
<td>3.13</td>
<td>Matching events by providing a function (\sigma_i) instead of messages.</td>
<td>38</td>
</tr>
</tbody>
</table>
3.14 Generated automaton for ambiguous specification “repeat trigger $m_1$ body end”.

3.15 Generated automaton for ambiguous statement “either trigger $m_1$ or trigger $m_2$ end”.

3.16 Generated automaton for unambiguous statement “either trigger $m_1$ or expect $e_1$ end”.

3.17 Generated automaton for statement “either expect (in, $\sigma_1$) or expect (in, $\sigma_2$) end”.

4.1 Components in a streaming application - CUT is TM.

4.2 Visualizing a data transfer lifecycle.
Listings

2.1 A Simple Grammar ................................................. 9
2.2 Example of a Context-Free Grammar (CFG) ....................... 10
3.1 CFG for our Domain-specific Language (DSL) ..................... 21
3.2 Productions for concatenating events .............................. 22
3.3 Union of executions ............................................... 23
3.4 Creating blocks within specifications .............................. 24
3.5 Specifying unordered set of events ............................... 28
3.6 Declaring constraints and requirements on blocks ............... 30
3.7 Mocking specific behavior of components ......................... 35
3.8 Inspecting internal state ......................................... 36
3.9 Specifying request responses in a LIFO manner ................. 39
4.1 A test specification in Java created using a builder pattern ... 47
4.2 Creating and Executing a Test Case .............................. 48
4.3 Use case test specification ....................................... 52
Acronyms

ε-NFA  Nondeterministic Finite Automaton with Epsilon Transitions. 12 29

API  Application Programming Interface. 52

CFG  Context-Free Grammar. 5 10 23 40 46 49

CSGs  Context-Sensitive Grammars. 41

CSP  Communicating Sequential Processes. 6

CUT  component under test. 21 22 23 29 30 32 36–39 46–50 52 54

DFA  Deterministic Finite Automaton. 10 12 24 26 31 44

DSL  Domain-specific Language. 5 10 23 26 28 39 42 44 46 48 49

EFSM  Extended Finite State Machine. 51 53

FSM  Finite State Machine. 51 53

LBA  Linear Bounded Automaton. 41

NFA  Nondeterministic Finite Automaton. 10 12 25 26 44

OOP  Object-Oriented Programming. 1 2 5

SUT  System Under Test. 5 6
Abstract

This thesis addresses the problem of unit testing components in message-passing systems. A message-passing system is one that comprises components communicating with each other solely via the exchange of messages.

Testing aids the developer in detecting and fixing potential errors and with unit testing in particular, the focus is on independently verifying the correctness of single components, such as functions and methods, in the system whose behavior is well understood. With the aid of unit testing frameworks such as those of the xUnit family, this process can not only be automated and done iteratively, but easily interleaved with the development process, facilitating rapid feedback and early detection of errors in the system.

However, such frameworks work in an imperative manner and as such, are unsuitable for verifying message-passing systems where the behavior of a component is encoded in its stream of exchanged messages.

In this work, we recognize that similar to streams of symbols in the field of formal languages and abstract machines, one can specify properties of a component’s message stream such that they form a language. Unit testing a component thus becomes the description of an automaton that recognizes such a specified language.

We propose a platform-independent language-recognition approach to creating unit testing frameworks for describing and verifying the behavior of message-passing components, and use this approach in creating a prototype implementation for the Kompics component model.

We show that this approach can be used to perform both black box and white box testing of components, and that it is easy to work with while preventing common mistakes in practice.

*Keywords* — unit testing, message-passing, verification
Chapter 1

Introduction

Even the simplest, non-trivial systems running in production fail occasionally — sometimes bringing about disastrous consequences like data loss and system-wide outages. With software becoming more and more ubiquitous in increasingly vital parts of our lives, their reliability become more important and today, software companies spend a large amount of resources in a quest to detect and fix errors in their developed systems.

Delivering high quality and reliable systems is no trivial task. The introduction of errors into the system can happen at any phase of the development process \[48\], for example, by the developers during the development phase or during requirements elicitation and specification and unfortunately, no technique or tool is able to detect all of these errors. Consequently, there continues to be research studying the variety of indicators and factors in the software development process that contribute to building more reliable systems and detecting more errors as quickly as possible \[54\] \[51\].

One universally accepted method for producing reliable software systems is through testing. It involves the execution of the system with the sole purpose of finding errors — verifying the differences between given input and expected output from the system \[46\]. As every error in the software contributes a potential failure of the system to perform correctly, the system is tested in many different scenarios in an attempt to verify all possible paths in the software program.

Testing can be performed at various levels depending on how much of, and at what granularity the system is to be verified. At the lowest level is unit testing \[23\], focusing on verifying the smallest testable components of the system — for example, functions or classes in plain Object-Oriented Programming (OOP) systems or actors, processes or components \[1\] in message-passing systems.

\[1\] we refer to a unit in a message-passing system as component throughout this paper
Several unit testing frameworks exist for systems written in OOP languages, offering a practical approach to increasing the quality of such systems [22][50]. They allow the tester to write unit tests that verify the correctness of, for example methods, by supplying input arguments and verifying the expected output through return values. Such frameworks facilitate the interleaving between testing and developing a program and enable rapid feedback during the development process especially since the unit tests are automated by the framework. Early detection of errors informs the developers about the vulnerable locations of the system, enabling them to more carefully test such error prone parts even more rigorously than others as desired.

A message-passing system is one where the components that comprise the system communicate with each other solely by exchanging messages. Consequently, unit testing a messaging component involves the verification of not just the component’s internal state but also its behavior which is encoded in the stream of exchanged messages. As a result, specifying tests that can be subsequently used for automation by frameworks is not a straightforward process as in the case of purely imperative or functional programs.

As an example, consider the programmer looking to verify the expected behavior of a currently developed component similar to the following:

"If I send a message $m_1$ to the component, I expect to see only the message $m_2$ leaving it, then sending the message $m_3$ to it should cause the emission of either message $m_4$ or $m_5$".

At specific states during the component’s execution, the programmer would like to send some messages into the component and at others, verify outgoing messages from the component. Attempting to use a unit testing framework of the xUnit [15] family to verify such a simple behavior would be just as cumbersome as it is error prone as they usually work by supplying input arguments to methods and making assertions against the returned values.

These frameworks do not provide primitive concepts of components, messages or behaviors to begin with and consequently can not readily provide mechanisms to support assertions on the actual act of receiving and emitting messages to and from the component at runtime. Thus simply sending a single message requires the programmer to write a large amount of boilerplate code, chaining method invocations between the testing framework and the message-passing framework. Also of note is the nondeterminism involved in the final expectation of either message $m_4$ or $m_5$, adding an extra layer of complexity to the components behavior. As a result, there is a need for testing frameworks that allow the programmer to specify the desired behavior of a messaging component in an execution-oriented manner similar to the example above.
1.1 Problem Statement

As the size or complexity of a system increases, so do the chances of introducing errors even more rapidly. Message-passing systems [36] are inherently complex and this is amplified when processes work independent of each other in a distributed setting, due to the inherent asynchrony and possibility of partial failure where subsets of processes may fail at any time in production. Fortunately, unit testing goes a long way in verifying these systems as it has been shown that even the most critical failures of such systems can be avoided using such tests [55].

Unfortunately, most tools available for unit testing allow the tester to describe tests in an imperative manner. While this way of testing may prove effective for their intended languages or frameworks, they are not sufficient for verifying the behavior of a message-passing component as such tests can not easily take into consideration, the streams of messages exchanged by the component as well as the relative ordering between the messages that make up the stream. Thus the question arises:

“What techniques can be used to facilitate iterative and automated unit testing of message-passing components?”

1.2 Contributions

The main contributions of this thesis are:

• A platform-independent, language-recognition approach to unit testing message-passing systems.

• A [DSL] for specifying test cases over a sequence of events in a similar manner to writing regular expressions over a sequence of characters.

• Mappings from the proposed [DSL] to automata.

• The design and architecture of a tool called KompicsTesting, based on the proposed approach and [DSL] as well as a use case study of the practicality of this tool.

1.3 Goals

The aim of this thesis project is to improve upon the current level of abstraction when testing message-passing systems in order to create high quality and reliable software built using this paradigm. A consequence of which might increase developer and company confidence in the delivered systems as well as customer satisfaction [54]. To carry out this improvement, this thesis project will be split into two separate subtasks. First and foremost is the evaluation and proposal of techniques that facilitate the detection of errors in such systems.
at early phases of the development process and in an automated manner that enables interleaving between testing and development activities. The next step is to verify the feasibility of the proposed techniques by developing a prototype implementing these techniques. Finally the prototype is evaluated with regards to its practicality.

1.4 Scope

The focus of this thesis is on the detection of errors within independent components of the system as is done with unit testing. Techniques for performing other granularity of tests such as system and integration tests are not explicitly explored. Thus, the proposed framework and implemented prototype will allow users to:

- Implement frameworks for unit testing messaging components in a given message-passing model and implementation; allowing developers to
- Easily detect errors during the development of single components, that would otherwise not be found until later stages.

1.5 Methodology

A literature study will be carried out in order to collect information on the current state of art in testing message-passing systems. This study forms the base of the thesis project, providing enough knowledge on current techniques and practices to complete further tasks.

Armed with knowledge from the literature study, a trial-and-error method will be used in creating a platform agnostic model for testing message-passing systems.

To evaluate the feasibility of the model created in the previous step, an experimental method will be used to create a prototype implementation of the proposed model. This method will produce a functional tool that can be evaluated in the further steps.

An illustrative case study is then carried out with respect to the practicality of the tool

1.6 Outline

This thesis is organized as follows: Chapter 2 presents the background containing an overview of the theory needed to follow this work. Chapter 3 presents our proposed platform independent framework for testing message-passing systems. In chapter 4, we evaluate the proposed framework, presenting a prototype implementation for the Kompics component model. Following an overview of related work (chapter 2.6), chapter 5 summarizes the thesis and discusses future work.
Chapter 2

Background

In this chapter, the background of the thesis is explained. It is divided into several sections. Section 2.1 discusses the need for automated testing while the following sections gives the necessary knowledge to understand our approach. Section 2.2 introduces concepts behind modelling messaging components as needed to follow this thesis. Section 2.3 and 2.4 presents formal languages and finite automata from language theory while section 2.5 highlights concepts from the Kompics component model needed to follow the design and implementation of our prototype.

2.1 Automated Software Testing

By testing, the programmer or tester exercises a system to verify that it does what it is implemented to do and conversely, that it does not do anything else [46]. This can be done at different levels ranging from unit testing through integration and acceptance testing [38].

With unit testing, a relatively small piece of code whose behavior is well understood by the tester is exercised. Usually this is a single function, method or class in [OOP] languages and in message-passing systems, a single actor, process or component. Although the behavior of such a unit is detailed, it is usually not possible to test every possible input and output combinations for non-trivial units as writing and executing all such test cases would demand too much time and resources to be economically feasible. Nevertheless, testers do want to unit test their systems as sufficiently as possible without devoting an unnecessary amount of time to it. Thus the appeal of performing automated unit testing [14], relieving the tester of manually executing every test case.

The tester initially provides a specification containing the expected behavior of the [System Under Test (SUT)] and a test framework verifies the correspondence of the SUT to the specification. Such verifications can subsequently be repeated any number of times at little to no extra cost to the tester.
The SUT can be tested using either white box or black box techniques depending on the assumptions made by the test logic. White box testing is a method of testing that assumes knowledge of and verifies the internal implementation of the SUT while black box testing makes no assumptions on the internal implementation of the SUT only verifying its functionality by injecting stimuli and expecting specified responses to and from the SUT. One may also write tests that combine both techniques.

2.2 Executions and the I/O Model

Message-passing systems typically are based on one of several mathematical models for concurrent and distributed event systems. For example Go follows the Communicating Sequential Processes (CSP) model while Erlang and Akka follow the actor model while Kompics is based on. Systems built on such models can thus be described by an input/output automaton model and this section describes applicable concepts and terminology associated with this model as appropriate to the scope of this thesis.

2.2.1 Events

Unlike systems that perform a set of computations according to their provided input and then terminate, components in message-passing systems are reactive entities that continuously perform events by receiving and reacting to input from their environment in addition to performing computations. In this model, events performed by a component can be categorized as either input, output or internal events where an input event is the reception of a message generated by the environment, output is the generation and transmission of a message to the environment and an internal event is the performance of a computation that may change the internal state of the component. The set of events S that can be performed by a component can thus be partitioned into three disjoint sets in(S), out(S) and int(S) respectively. We abuse this notation by saying that the set in(S) contains events of the form in(ei), out(S) contains events of the form out(ei) etc, where ei is an event. The set of events together with its partitions form the interface between a component and its environment. We note here that the set ext(S) = in(S) ∪ out(S) exclusively concerns interactions between the component and its environment and are thus called external events.

2.2.2 Executions

Components in message-passing systems execute in steps and in each step, exactly one event is performed. When the component runs, its execution can be denoted by a sequence of steps where in the first step, the first event e1 is performed by the component at runtime, e2 is the event for the second step and so on. A sub-execution is a subsequence of such an execution.
For example, the sequence \( E_1 = (\text{in}(m_1), \text{int}(i_1), \text{in}(m_2), \text{out}(m_3), \ldots) \) represents an execution of some component \( c \). The first event performed by \( c \) is the reception of a message \( m_1 \), then the performance of some internal event \( \text{int}(i_1) \), next it receives another message \( m_2 \) before sending a message \( m_3 \) and so on. This execution represents an infinite execution (denoted by ellipsis) which is possible because, as mentioned, components may continuously perform events.

We may also be interested in the state changes of a component during execution as each performed event changes the component state according to its logic. For such occasions we denote an execution by alternating each performed event with the new component state. Thus in the previous example, the execution will become \( E_2 = (\text{in}(m_1), s_1, \text{int}(i_1), s_2, \text{in}(m_2), s_3, \text{out}(m_3), s_4, \ldots) \).

Finally, whenever the actual event direction (\( \text{in}, \text{out}, \text{int} \)) is not needed we simply denote an execution using variables. The previous example may be rewritten as \( E_3 = (e_1, s_1, e_2, s_2, e_3, s_3, e_4, s_4, \ldots) \).

### 2.2.3 Behaviors

The behavior of an execution sequence \( E_i \) is a subsequence \( \gamma \) of \( E_i \) consisting only of external events \([32]\). Since the external events only concern the interaction between the component and its environment, the behavior denotes the portion of an execution that is observable by the external environment when executing the component and will be the primary basis through which we perform black box testing in our approach.

### 2.3 Formal Languages

#### 2.3.1 Formal Languages

A formal language \([35]\) is a set of sequences of symbols formed according to some specified rule. Such a language has an alphabet — a set of valid symbols for constructing a sequence in that language. For example, the set \( \{ab, ac\} \) constitutes a language containing exactly two sequences \( ab \) and \( ac \) while the alphabet of the language is the set of characters \( \{a, b, c\} \). The rule for this language could be specified simply as “every sequence must begin with an \( a \) and followed by exactly one \( b \) or \( c \)”.

It is rarely feasible to describe a language as we just did by listing out all the sequences that are members of the language, nor is it particularly desirable to verbally describe all possible languages. Instead, one may simply give a formation rule that describes the intended language. This may then be used either to generate members of that language or check that a given sequence belongs to the language. Such a rule may be a grammar \([11]\) or an automaton \([21]\) with each rule having several variants for different classes of languages. For
the scope of this thesis we will only discuss regular and context-free grammars and finite automata.

Kleene Closure

We start by defining the Kleene closure, or closure on a set of symbols [13][21]. Let \( \Sigma \) be a finite set of symbols. The closure on the set \( \Sigma \), denoted \( \Sigma^* \), is the set of sequences that can be formed by taking any number of symbols in \( \Sigma \), with repetitions allowed. For example if \( \Sigma = \{a, b\} \), the closure \( \Sigma^* = \{a, ab, aa, ab, aba, ...\} \). The closure on any set always includes the empty sequence, denoted \( \varepsilon \) since one may choose to take zero number of symbols.

2.3.2 Grammars

A grammar \( G \) is formally described as a four-tuple \((N, \Sigma, P, S)\) where \( N \) is a set of nonterminal symbols, \( \Sigma \) is the alphabet containing terminal symbols, \( P \) a set of productions and \( S \in N \) a start symbol [21].

A production \( p \in P \) is of the form \((\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*\) where \( ^* \) is the closure operator — hence the right hand side may be empty. Thus a production may have at least one nonterminal symbol on the left-hand side of the arrow while the right-hand side may contain any number of terminals and nonterminals.

Generating a Sequence using a Grammar

If the grammar \( G \) generates exactly the set of sequences that make up some language \( L \), then we say that \( G \) describes \( L \) and write \( L(G) = L \). To generate a sequence in \( L \) using \( G \), we start with the start symbol \( S \) of \( G \) and expand using the right-hand side of any production of \( S \), replacing the expanded symbol with its right-hand side. We repeatedly do this until no nonterminal symbols are left in the expanded sequence.

As an example, consider the simple grammar \( G = (N, \Sigma, P, S) \) whose productions \( P \) are shown in listing 2.1. Nonterminal symbols are denoted with capital letters while terminal symbols are denoted using small letters. Thus \( N = \{A, B\} \), the alphabet \( \Sigma = \{a, b, c, d\} \) and \( S \) is the start symbol. We begin with the start symbol and select the production on line 1 to expand the sequence. Thus replacing \( S \) with the sequence \( ABc \). Next we do the same for a remaining nonterminal \( A \), expanding it with \( a \) to give \( aBc \) and finally, expanding \( B \), using the production on line 4 which gives us the final sequence containing the terminals \( abc \). Note that we could have started by expanding \( S \) using the production on line 2 which would have given us the final string \( bcd \) instead.
2.3.3 Regular Grammars

Regular grammars restrict the form of the productions that they can contain. Here, the left-hand side must contain exactly one nonterminal symbol while the right-hand side may contain a single terminal optionally followed by a nonterminal symbol, or a nonterminal symbol followed by a terminal symbol. The former case is called a *right regular* grammar while the latter case is called a *left regular grammar*. The right-hand side of a regular grammar may also be empty. A language described by a regular grammar is called a *regular language*.

2.3.4 Regular Expressions

We can forego the use of regular grammars in place of an equivalent *regular expressions* notation for describing regular languages in a more concise manner [29]. A single regular expression describes a language that is regular. The basis for regular expressions are the symbols of the language’s alphabet $\Sigma$ such that if $a \in \Sigma$, then $a$ is also a regular expression and $L(a) = \{a\}$ — that is the language described by a terminal symbol consists of exactly the single symbol.

Regular expressions can be built from smaller regular expressions using the following operations that are closed on regular languages [21].

**Union** The union of two languages $L$ and $M$, denoted $L \cup M$ is the language containing all sequences that are either in $L$ or in $M$ or in both. For example, if $L = \{ac\}$ and $M = \{ac, dc\}$, then the union of both languages $L \cup M = \{ac, dc\}$. Hence if $R$ and $S$ are regular expressions, then $R | S$ is also a regular expression describing the language $L(R) \cup L(S)$. The operator $|$ denotes the union of two regular expressions.

**Concatenation** The concatenation of two languages $L$ and $M$, denoted $LM$ is the language containing all sequences that are formed by taking a sequence in $L$ and appending a sequence in $M$ to it. Using the previous example, the concatenation of both languages $LM = \{acde\}$. Hence if $R$ and $S$ are regular expressions, then $RS$ is also a regular expression describing the language $L(R)L(S)$.

**Kleene Closure**. As described in section 2.3.1, the closure on a language $L$, denoted $L^*$ is the language containing all sequences formed by taking any number of sequences (with repetitions) from $L$ and concatenating them. Using the previous example, $L^* = \{ac, acac, acacac, \ldots\}$.
2.3.5 Context-free Grammars

Compared to regular grammars, a CFG is restrictive on the form of the productions that it can contain — the left-hand side of every production must contain a single nonterminal symbol. They can thus be used to describe a larger set of languages than regular grammars. Listing 2.2 shows an example of a CFG. A language described by a CFG is called a context-free language. Pushdown automata are finite automata that recognize a subset of all context-free languages. Essentially they are finite automata with access to an unbounded stack on which they can perform operations such as push and pop input symbols. A typical application of a CFG is in the description of programming languages as those form a subset of the context-free languages. We will also make use of this convenience when describing a DSL later in this thesis.

2.4 Finite Automata

A finite state automaton (from hereon automaton) is an abstract machine made up of states and transitions which connect any two states and are labelled with an input symbol.

An automaton can be in any such state at a given time, changing states by following transitions. It can be used to create a function \( M : \Sigma^* \rightarrow B \) (where \( B = \{ \text{true}, \text{false} \} \)) that returns true or false depending on whether or not it recognizes or accepts a given sequence \( s \in \Sigma^* \). This is determined by consuming the input symbols in \( s \), following transitions labelled by that input and returning true if after consuming all input symbols in \( s \), the automaton is in a final or accepting state — we say that the automaton accepts the sequence and rejects is otherwise.

An automaton can either be a Deterministic Finite Automaton (DFA) or a Nondeterministic Finite Automaton (NFA). A DFA has exactly one destination state for any given state and input symbol pair while an NFA may have several possible destination states for any such pair. As a result, a NFA may be in several states at once.

2.4.1 Deterministic Finite Automata

Formally, a DFA \( M \) is defined as the five-tuple \( (Q, \Sigma, \delta, q_0, F) \) where \( Q \) is a finite set of states, \( \Sigma \) is a set of input symbols, \( \delta \) is a transition function \( \delta : \Sigma \times Q \rightarrow Q \)

Listing 2.2: Example of a CFG

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>→</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>→</td>
<td>aBC</td>
</tr>
<tr>
<td>C</td>
<td>→</td>
<td>c</td>
</tr>
<tr>
<td>D</td>
<td>→</td>
<td>Cd</td>
</tr>
</tbody>
</table>
mapping a state and symbol pair to a new state, $q_0 \in Q$ is a start state which is the initial state of the automaton, $F \subseteq Q$ is a set of accepting states.

Instead of explicitly listing out the mappings of the transition function $\delta$ for a given automaton, we can visually describe it using a state diagram as shown in figure 2.1 Here, $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, c, t\}$ and $F = \{q_3\}$. Transitions are arrows labelled by an input symbol. The start state $q_0$ has no incoming transitions while there is only one final state $q_3$, depicted by a double-bordered circle.

![Figure 2.1: A DFA recognizing the sequence “cat”](image)

A DFA can be used to recognize a sequence of symbols. The language $L(M)$ of a DFA $M$ is the set of all sequences recognized by $M$, we also say that $M$ describes the language $L(M)$.

As an example, figure 2.1 shows a DFA that recognizes the sequence cat. A simulation of this DFA starts at state $q_0$ and tries to consume the first symbol c, hence from $q_0$ it follows the transition labelled c to state $q_1$, consuming c successfully. Now the remaining sequence is at so it does the same for the next symbol in the sequence a and so on until no more symbols are left.

Note that we have not specified transitions for every other symbol at each state, and the DFA must have exactly one valid transition for each state and input pair. We only show those transitions that lead to an accepting state and imply that all other non-specified transitions point to an implicit error state such that the automaton rejects the sequence. Finally we note that, like for regular grammars and regular expressions, the language $L(M)$ of a DFA $M$ is also a regular language. Thus if a language can be described by a regular grammar, then it can also be described by a DFA and vice versa [10] [21].

### 2.4.2 Nondeterministic Finite Automata

A NFA can be in several states at the same time since it may have multiple transitions for a given input symbol and state pair. The automaton is said to guess its next state by following all eligible transitions.

Formally, a NFA $M$ is defined as the five-tuple $(Q, \Sigma, \delta, q_0, F)$ where $Q$ is a finite set of states, $\Sigma$ is a finite set of input symbols, $q_0 \in Q$ is the start state, $F \subseteq Q$ is a set of accepting states and $\delta$ is a transition function $\delta : \Sigma \times Q \rightarrow \Gamma$ mapping each state and input symbol pair to a set of states $\Gamma \subseteq Q$ [21]. Thus, the transition function alone marks the difference between a NFA and a DFA.
2.4.3 NFAs with Epsilon-Transitions

A Nondeterministic Finite Automaton with Epsilon Transitions (ε-NFA) with epsilon transitions is an extended NFA with the single additional ability to allow a transition without consuming any input string. Thus an ε-NFA can have unlabelled transitions called ε-transitions, allowing it to make a spontaneous move to the next state. Figure 2.3 shows an example of an ε-NFA. In a state diagram unlabelled transitions are marked with the special ε symbol only as a visual convenience as the symbol does not belong to the input alphabet of the automaton. ε-NFAs do not extend the class of describable languages defined by NFAs or DFAs or regular grammars — they all define regular languages. In fact, any given ε-NFA can be converted to an equivalent NFA. However, they offer more illustrative and programming convenience and will be used later in this thesis.
2.4.4 Concatenating Finite Automata

Since the described finite automata all define regular languages similar to regular grammars and regular expressions, the union, concatenation and closure operations introduced in section 2.3.4 can be performed on any such automaton. Here we describe the construction of an automaton $MN$ from the concatenation of any two automata $M$ and $N$ as presented in [21]. This operation is implied throughout this thesis and as such is presented here for reference. A more specialized description for the union and closure operations on automata is illustrated in sections 3.3.2 and 3.4.3 respectively.

To construct an automaton $MN = (Q, \Sigma, \delta, q_0, F)$ from two automata $M$ and $N$ where $M = (Q_M, \Sigma_M, \delta_M, q_{0M}, F_M)$ and $N = (Q_N, \Sigma_N, \delta_N, q_{0N}, F_N)$, we set $Q = Q_M \cup Q_N$, next we set the start state of the first automaton $q_{0M}$ as the start state $q_{0}$ of $MN$ and the accepting states $F_N$ of the second automaton as the accepting states $F$ of $MN$. The transition function $\delta$ uses the mappings from $\delta_M$ and $\delta_N$ with additional $\varepsilon$-transitions from each final state $q \in F_M$ in the first automaton to the start state $q_{0N}$ of the second automaton.

The idea here is that the first part of recognized sequence is delegated to the automaton $M$ and once this part is recognized, an $\varepsilon$-transition takes the automaton $MN$ to the start state $q_{0N}$ of $N$, where the second part of the sequence takes the automaton to an accepting state.

An example of this is shown in figure 2.6 as an automaton constructed from the automata in figure 2.4 and 2.5.

Figure 2.6: Concatenating automata in figures 2.4 with 2.5 recognizing the sequence “acdc”.

\[
\text{start} \rightarrow q_0M \xrightarrow{a} q_1M \xrightarrow{c} q_2M
\]

\[
\text{start} \rightarrow q_0N \xrightarrow{d} q_1N \xrightarrow{c} q_2N
\]
2.5 The Kompics Component Model

Kompics is a component based, message-passing model for building distributed systems. Components in Kompics are event-driven entities that communicate by exchanging messages (in the form of events) with each other. Events are simply data-carrying objects in the system. Components provide communication interfaces via bidirectional ports and are connected to each other via channels binding any two ports. The following sections describe the key primitives and concepts as required to follow this thesis — more details can be found here [6].

2.5.1 Ports

Ports in Kompics embody the interface between a component and its environment. They are bidirectional entities through which events are sent to and received from a component. Through ports, Kompics provides a type system for events within the system. Unlike systems like Akka [53] and Erlang [7] where messages may be addressed to any component, ports define the events that may go in and out of each component.

A port has two directions which we label positive and negative as well as a port type that declares a set of event types that are allowed to pass through it in each direction of the port. We denote a port \( \alpha = (+\alpha, -\alpha) \) where \( +\alpha \) is the positive direction and \( -\alpha \) is the negative direction and say that a port \( \alpha \) allows an event \( e \) in some direction \( d \) if its port type declares the type of \( e \) in direction \( d \).

As communication interfaces, we can think of a port \( \alpha = (+\alpha, -\alpha) \) as a service interface and associate requests with its negative side \( -\alpha \) and responses with its positive side \( +\alpha \). Thus a component that supplies service \( \alpha \) declares \( \alpha \) so that request events are received, incoming from \( -\alpha \) while response events are outgoing from \( +\alpha \). We also say that the component declaring a port in this manner provides the port. Conversely, a component that consumes service \( \alpha \) declares \( \alpha \) so that responses are incoming from \( +\alpha \) while requests are outgoing from \( -\alpha \). We say that the component requires the port.

Again, we label the two sides of a port from the perspective of a component that declares the port by saying that the side of the port emitting incoming events to the component is the inside port while the side emitting outgoing events is the outside port. Thus for a component that provides port \( \alpha \), the inside and outside ports are \( -\alpha \) and \( +\alpha \) respectively while conversely, for a component that requires \( \alpha \), they are \( +\alpha \) and \( -\alpha \) respectively. Finally we say a component triggers an event on a port if it sends an event through that port — this is typically done from the inside, going out.
Port `pingpong` allows `ping` and `pong` events in the negative and positive directions respectively. A providing component (ponger) has a negative inside port and a positive outside port, while a requiring component (pinger) has a positive inside port and a negative outside port. Channels connect ports of opposite directions, (here + and −).

As an example, consider a system where clients send `ping` requests and servers reply with `pong` responses — we call them `pingers` and `pongers` respectively. To implement such an protocol we create a `pingpong` port with its port type declaring `pong` events in the positive direction and `ping` events in the negative direction. This is shown in figure 2.7. Now, being a service provider, a ponger component provides a `pingpong` port while a pinger component requires a `pingpong` port as shown in figure 2.8. Thus a pinger component may emit `ping` requests from its outside port while a ponger component emits `pong` responses from its outside port. Conversely, note that the pinger has its positive port as its inside port, receiving `pong` events as incoming responses while the ponger has its negative port as its inside port, receiving `ping` events as incoming requests.

2.5.2 Channels

Channels create connections between two components via their declared ports. They can be thought of as bidirectional communication pipes that carry events from one port to another. However, connections are only possible for any two ports of the same port type and having opposite directions — that is given two ports α = (+α, −α) and β = (+β, −β), only the pairs of port (+α, −β) and (−α, +β) can be connected by channels. A channel is depicted in figure 2.8 using a bar connecting both ports. Here the channel connects the outside port...
Figure 2.9: Event handlers are depicted with rounded rectangles. Here handler handlePong is subscribed to port pingpong. Outgoing events are triggered on inside ports and depicted with dash-arrows and diamonds.

pingpong ports are declared by both components. Since one provides and the other requires the port, their outside ports have opposite directions.

Events are forwarded through a single channel in first in, first out (FIFO) order \[18\] — delivered to the destination port in the order that they were triggered at the source port. An event triggered on a port is broadcast on all channels connected to that port. Thus Kompics does not provide a mechanism for addressing events to specific components in the system. On arrival of an event at a destination port, it is queued up on that port until the component that declared the port is scheduled to execute that particular event.

### 2.5.3 Event Handlers

An event handler, or *handler*, in Kompics is a user-defined function for a component. A handler accepts events of a particular type and any of its subtypes (in the strongly typed programming language sense). Handlers are registered on ports and are executed whenever the component receives an acceptable event. We say that a registered handler for a given port is *subscribed* to that port.

Figure 2.9 shows a pinger component from the previous example having subscribed a handler on its required pingpong port. Such a handler is depicted by a rounded rectangle inside the component with an arrow from the subscribed port to it denoting the flow of events (incoming). As mentions previously, a component triggers an event from its inside port going out, this is depicted by a dash-arrow from within the component to the inside port, denoting the flow of events (outgoing).

Kompics guarantees a sequential execution of the handlers of a single component, preventing the need for internal state synchronization between handlers of that component. Handlers belonging to different components on the other hand, may be scheduled and executed concurrently.
2.5.4 Components

Components in Kompics are reactive entities which communicate asynchronously with each other by exchanging messages. Similar to actors in actor based systems [1], a component has some internal state associated with it as well as a message queue at its declared ports. It also has a set of event handlers which as mentioned are subscribed on its declared ports, and executed whenever some accepted event is received on that port. Handlers are the primary means by which a component updates its internal state.

A component can be encapsulated within another component, using parent-child relationships that form a component tree hierarchy, with a single root component `main`, that is initially started at runtime. The relationship between a parent component and its children components enables a flexible architecture for managing system complexity as well as the delegation of a component’s configuration to that component’s parent. For example, on creating and starting a component, the children components (or sub-components) are recursively created and started. It thus becomes the responsibility of the parent to bootstrap its sub-components, for example by setting up their communication channels.

2.6 Related Work

Recently there has been increasing interest in techniques for improving the reliability of message-passing systems. The most common being the use of imperative unit testing techniques and tools based on xUnit [15] where the focus is on testing the system by inspecting field members of objects and performing assertions on the output of functions after being called with a predefined input.

The Akka TestKit tool [2], provides a platform for performing unit and integration testing on actor systems based on the Akka framework [53], at various level of granularity. It allows the user to test that incoming sequences of messages are processed correctly, even in the face of nondeterminism that causes reordering of messages. However, such tests can only be performed using other actors to generate the stream of messages and listen for outgoing messages, unlike our approach that allows an interactive mechanism for generating streams of events. Akka TestKit, also does not use any language or automata based approach, nor does it require the tester to explicitly describe the expected behaviour of the actor.

Techniques based on formal methods have also been applied to the verification of message-passing systems. Formal specification languages like TLA+ [29] provide a proof system for modelling and verifying concurrent systems based on a provided specification of the system’s behavioral properies. However, such tools require a user to have advanced mathematical background and theorem proving skills, limiting its reach throughout software practitioners in the industry [39] [19]. Model checking is another formal method used to verify the conformance of a given model of a system to its intended specification. Most
model checking methods use state-space exploration [12] and can instil more confidence in a system by automatically enumerating and exhaustively exploring the system’s state-space to find errors. If all paths in the state-space have been successfully verified, then the system is said to be correct. However, state-space exploration can be an expensive process and in concurrent systems, it is not uncommon that it requires an exponentially larger or infinite number of states to be generated — a problem known as state-space explosion. Techniques such as partial-order reduction [34, 9] and dynamic partial-order reduction [31] exist to try and mitigate this problem by reduction of the number of explored states.

Previously, model checking methods have been applied to testing of message-passing systems. Systems like Basset [30], Jacco [56], and McErlang [16] have been developed specifically for actor systems. Basset provides a model checker for actor systems that compile to Java bytecode, built on top the Java PathFinder JPF [52], a more general model checker for Java programs. In order to efficiently explore an actor system, Basset avoids exploring low-level and library code, concentrating only on message scheduling related errors instead. It uses dynamic partial-order reduction as to prune the number of explored states. However, as mentioned in the Jacco paper [56], its message scheduling assumptions lead to false negatives in generating explorable states which leads to large state spaces. Jacco improves on Basset by implementing a new scheduling approach and technique for reducing the state space of the actor system. Additionally, its architecture abandons the reliance on the JPF platform leading to better performance in model checking time. McErlang is a model checker for Erlang programs. It allows the user to encode correctness properties of the system as automata specified in Erlang while the model checker checks the system against the specified properties.
Chapter 3

A Framework for Testing Message-Passing Systems

3.1 The Language of Event Streams

In accordance with the goal of unit testing, a programmer would like to assert assumptions against the behavior of some [CUT] component under test and its interaction with the environment — the aim being to increase the confidence in the component’s implementation. Consequently, the programmer provides a test specification or specification containing assumptions of the expected behavior while the test framework verifies that the [CUT]'s behavior corresponds to the specification. This raises the question of what the contents of such a specification should be in the context of message-passing systems and how it could assist a test framework in the verification process at test execution time.

Consider the following expected behavior of a [CUT] similar to chapter 1. "If I send a message $m_1$ to the component, I expect to see only the message $m_2$ leaving it, then sending the message $m_3$ to it should cause the emission of one or more messages $m_4$".

We can consider this as a specification, providing a test framework a verifiable description of the expected behavior of the [CUT] — in other words, the programmer describes a number of correct executions of a [CUT] while the framework verifies that the [CUT]'s observed behavior matches one of the specified executions. Thus we say that this specification $S$ describes an execution set $E$ — a set of execution sequences. In this case the execution being described can also be concisely written as $E = (in(m_1)out(m_2)in(m_3)out(m_4)out(m_4)^*)$ where $^*$ is the Kleene operator denoting zero or more occurrences of the outgoing message $m_4$. As shown in section 2.2, we can assign variables to these events and write $E = (e_1e_2e_3e_4e_4^*)$ where $e_1 = in(m_1), e_2 = out(m_2)$ and so on. Thus
the programmer actually wants to describe the same set of events as the regular expression $e_1e_2e_3e_4^*$ where the alphabet of the language contains the set of symbols $\Sigma = \{e_1, e_2, e_3, e_4\}$. As a result, it can be seen that the execution set described by such a specification $S$ forms a regular language, allowing us to create a finite automaton, generated from $S$ and recognizes exactly the described set of correct executions.

For the verification process, the test framework can then execute an instance of the CUT observing the events that occur at runtime (or more strictly, test execution time) and use them as the constituents of the execution sequence (input symbols) to simulate the constructed automaton. For our example, an automaton $M_E$ can thus be created as shown in figure 3.1 so that on observing event $e_i$ at runtime, we either transition to the next state pointed at by the transition labelled $e_i$ or fail the test case immediately if there are no such transitions. Hence, the test case would be considered successful only if the execution reaches state $q_4$ and no other events are observed.

Figure 3.1: An automaton recognizing the same language as $e_1e_2e_3e_4^*$.

Note that in this particular test case, we have not referred to the internal state of the CUT. If we did, then we would have been performing white box testing and our described execution would have been of the form $E = e_1s_1e_2s_2e_3s_3e_4s_4\ldots$ where $s_i$ represents the internal state of the CUT after performing event $e_i$ (see section 2.2). Thus in this test case we are strictly performing black box testing.

**Online vs Offline Recognition:** As we have mentioned, the observed events at runtime form a sequence that is used to simulate the constructed automaton. This process of checking whether or not the automaton recognizes the observed sequence can either be performed on the fly as events are observed by the framework, or after all the events of the sequence have been observed and recorded. We call the former method online recognition and the latter offline recognition of the observed sequence. Both techniques have their advantages and disadvantages. For example, when performing online recognition, it becomes difficult to have the automaton accurately decide how long to wait for events to occur [3.10] or in some cases, when all events have in fact occurred [3.4.3]. On the other hand, with offline recognition, we are unable to easily specify test cases that require interactions [3.5.3] and it places extra storage requirements since the entire execution sequence must initially be recorded. Another advantage of online recognition is that we do not need to continue to run the automaton.
if the observed sequence will lead to a failed test case. For example, in the
previous example using the automaton in figure 3.1, on observing the first event
e_i \ne e_1, the test case fails immediately without waiting for subsequent events to
occur but with offline recognition this can not be detected until all events have
occurred. Throughout this thesis, we make use of online recognition.

3.2 A DSL for Writing Specifications

Although we have shown how a programmer may describe executions that form
a regular language, the symbols of such a language are events (directions and
message pairs) and not characters. Hence some mechanism is needed for writing
test specifications. In the latter case where the symbols are characters, one may
simply use a regular expression matcher, possibly provided by the program-
ing language or platform but this is not possible in the former case and more
importantly is the fact that we do not want the describable language of our
specification to be restricted to regular languages. Our approach implements a
mechanism for specifying execution sequences as, primarily but not exclusively,
regular languages while facilitating interactive and non-deterministic testing.
The advantage of this is that it allows the programmer to utilize likely familiar
techniques and concepts from regular expressions when writing tests.

In the following sections, using the CFG (with start symbol S) shown in
listing 3.1 we present a DSL for writing such test specifications that can sub-
csequently be converted into an automaton and instructions to be executed by a
test framework.

$$
\begin{align*}
S & \rightarrow \text{Exec} \\
\text{Exec} & \rightarrow \text{repeat } n \ ? \ \text{Hdr } \text{body } \text{Body } \text{end} \\
\text{Hdr} & \rightarrow \text{allow } e_i^+ \mid \text{disallow } e_i^+ \mid \text{drop } e_i^+ \\
& \mid \text{blockExpect } e_i^+ \\
\text{Body} & \rightarrow \text{expect } \text{Event}^+ \mid \text{either } \text{Body}^+ \text{ or } \text{Body}^+ \text{ end} \\
& \mid \text{Exec} \mid \text{trigger } m_i^+ \mid \text{inspect } \alpha \\
\text{Event} & \rightarrow e_i \mid \text{unordered } e_i^+ \text{ end}
\end{align*}
$$

Listing 3.1: CFG for our DSL.

3.3 Regular Tests

In this section, we illustrate the variants of regular expression operations offered
by our approach for describing executions and building larger executions from
smaller ones. In particular, we illustrate the union and concatenation operations
on languages described by a specification while the Kleene closure operation is
explained in section 3.4.3 when the concept of blocks have been introduced.
3.3.1 Concatenating Executions

Listing 3.2 shows the productions for concatenating a sequence of single events using the expect keyword. The symbol $+$ means one or more occurrences. Each statement that appears on the right-hand side of the Body non-terminal describes a unique language (set of executions) over the set of event alphabet and these languages are concatenated in their described order to form the language defined by the body of the test specification. As with the symbols of regular expressions, a single event $e$ matches itself and the language $L(e)$ consequently consists only of itself — that is $L(e) = \{e\}$.

In terms of the automaton created, at runtime the statement `expect e` would cause the automaton to move from the start state to the next and final state only when event $e$ has been observed — thus matching $e$. Observing any other event at this state leads to a failed test case. Figure 3.2 shows a DFA $M_e$ that recognizes the single event $e$.

To concatenate execution sets, we use the definition of the concatenation of regular languages as defined in sections 2.3.4 and 2.4.4. Using the `expect` keyword, a programmer can describe a sequence of events such that the language described is the concatenation of these individual events.

As an example, the statement “`expect $e_1e_2e_3$`” describes the language $L = \{e_1e_2e_3\}$ formed by concatenating the three languages $L(e_1) = \{e_1\}$, $L(e_2) = \{e_2\}$ and $L(e_3) = \{e_3\}$. Generally, given a statement $S = \text{"expect } e_1e_2e_3\ldots e_n\text{"}$, where each $e_i$ describes a unique language $L(e_i)$, an automaton $M_S$ recognizing $L(S)$ is created by sequentially concatenating each automaton $M_{e_i}$ that recognizes $L(e_i)$. Figure 3.3 shows an automaton constructed in this manner.

---

**Listing 3.2: Productions for concatenating events**

\[
\begin{align*}
\text{Body} & \rightarrow \text{expect Event}^+ \\
\text{Event} & \rightarrow e_i
\end{align*}
\]

**Figure 3.2: An automaton recognizing a single event $e$.**

**Figure 3.3: Automaton for recognizing the concatenation of events.**
3.3.2 Union of Executions

Listing 3.3 illustrates the constructs for creating the union of execution sets using the **either-or** conditional statement. As each **Body** nonterminal describes a unique language, the conditional statement contains two independent languages from the **either** and **or** branches, which are subsequently combined via the union operation on regular languages (see section 2.3.4) to form the described language of the conditional statement.

Conditional statements allow the programmer to describe possible paths within the state-space of the component, exactly one of which will be traversed depending on the observed events at runtime. These statements are convenient in situations where there are several alternative and possibly equivalent paths outgoing from a certain state of the CUT. In some cases, it may be inconvenient or difficult to reproduce a test environment that consistently guides the test case through a desired path while in other cases, the paths may be supplied to increase the robustness of the test case — the test case may be designed so that a random path is traversed at each test execution. Using the **Body** nonterminal to describe the branches, any statements are allowed within a conditional statement, including other conditional statements.

Given a conditional statement $S$ with statements $A$ and $B$ as specified by its **either** and **or** branches respectively, the language $L(S)$ described by $S$ is defined to be the union of the sets described by both branches — that is $L(S) = L(A) \cup L(B)$. Consequently, an automaton $M_S$ recognizing $L(S)$ would be equivalent to the automaton $M_{A\cup B}$ that recognizes the language $L(A) \cup L(B)$.

We construct a NFA for this purpose by combining sub-automata $M_A$ and $M_B$ for $L(A)$ and $L(B)$ respectively alongside each other as described in [21]. All states and transitions of the sub-automata remain throughout the construction. A new start state $q_{A\cup B}$ of $M_S$ is created by merging the start states $q_A$ and $q_B$ of $M_A$ and $M_B$ respectively. This new state will have the same outgoing transitions as the combined start states, allowing the NFA to reach the final states of either sub-automaton when verifying events at runtime. The set of final states of the NFA is the union of the final states of both sub-automata since any execution that is accepted by either sub-automaton is accepted by the overall NFA.

As an example, consider the statement $S = \text{"either expect } e_1 e_2 \text{or expect } e_1 e_3 \text{ end"}$. Figure 3.4 shows the constructed NFA $M_S$ for this statement. The start state $q_{A\cup B}$ contains the same transitions outgoing from the start states $q_{A0}$ and $q_{B0}$ of the sub-automata while the final states $q_{A2}$ and $q_{B2}$ of the NFA are those
3.4 Extending Regular Tests

So far we have discussed variants of regular expression operations (with the exception of the closure operation) that allow the programmer to describe executions that form a regular language. In this section, we describe constructs of our DSL that enables the description of a wider range of test scenarios while writing more concise and understandable specifications. Section 3.4.1 introduces the concept of blocks as a way to describe executions in units within a specification while sections 3.4.2 and 3.4.3 describe the two distinct types of blocks provided by the DSL with the latter being used to describe the Kleene closure of an execution. Section 3.4.4 illustrates constructs for describing non-deterministic executions where the ordering of events are either not important or unpredictable at specification time. Finally section 3.4.5 illustrates mechanisms for describing an even wider range of nondeterministic scenarios using blocks as well as placing requirements on blocks in order for a test case to be successful.

3.4.1 Blocks

The DSL provided by our framework is a block-structured language. This means that the language allows for the creation of blocks as well as nested blocks. In our case, it enables a programmer to group a sub-sequence of events (sub-execution) into a single unit.

A block is the topmost construct of a test specification and is shown in listing 3.4. It describes an execution which in turn may be made up of smaller sub-
executions using nested blocks. Here, \( n \) is a positive number while \(?\) means that it is optional. Every specified event belongs to a single block. As with a lot of block-structured programming languages, the benefits are manifold — a scope can be invoked throughout a block, facilitating the declaration of constraints and requirements. For example in a general purpose language such as Java, the visibility of variables may be constrained to a single block. In our case these constraints may be declared on a block so that they only affect the block’s described sequence at runtime.

As an example of how a requirement can be declared on a block’s described sequence, consider a programmer wanting to declare that a particular event \( e_0 \) must occur within a subexecution \( E_S = e_2e_3 \) of the execution \( E = e_1e_2e_3e_4 \). Note that the exact position of \( e_0 \) within \( E_S \) isn’t specified as it may not be predictable. Since all events must belong to a block, we assume that all events in execution \( E \) initially belong to some block \( b_E \). Now, the programmer may declare that the sub-sequence \( E_S \) is associated with a nested block \( b_{ES} \) so that events \( e_1 \) and \( e_4 \) continue to be associated with the outer block \( b_E \) while events \( e_2 \) and \( e_3 \) are associated with \( b_{ES} \). Now the programmer can declare that \( e_0 \) must occur at some point within \( b_{ES} \). Thus the actual described sequence becomes \( (e_1e_0e_2e_3e_4|e_1e_0e_3e_4|e_1e_2e_3e_0e_4) \). This particular technique is explained further in section 3.4.5.

The Language Described by a Block

The production in listing 3.4 splits a block statement into an optional header and a body section identified by the Hdr and Body nonterminals. The nonterminal Hdr generates statements that declare constraints and requirements on the events observed within the block as well as any nested blocks while the Body as highlighted so far generates statements that describe execution sets belonging to the block. The language described by a block is the set of execution sequences formed by applying the specified constraints and requirements declared in the block header to the execution sequences described by the block body and subsequently applying a block operation as determined by the block’s type. The next two sections 3.4.2 and 3.4.3 describe the two types of blocks and the implied operation used to construct their described language.

3.4.2 Repeating Executions

Consider the example specification \( S_0 \) from 3.1 repeated here for convenience:

"If I send a message \( m_1 \) to the component, I expect to see only the message \( m_2 \) leaving it, then sending the message \( m_3 \) to it should cause the emission of one or more messages \( m_4 \)."

and the following specification \( S_1 \):
"If I send a message \( m_1 \) to the component, I expect to see only the message \( m_2 \) leaving it, then sending the message \( m_3 \) to it should cause the emission of one or exactly 4 messages \( m_4 \)."

The specification \( S_0 \) we know can be written as \( S_0 = e_1 e_2 e_3 e_4^* \) while \( S_1 \) can be written as \( S_1 = e_1 e_2 e_3 e_4 e_4 e_4 e_4 e_4 e_4 e_4 e_4 e_4 e_4 e_4 \). However note how the closure notation * makes specification \( S_0 \) a lot more concise as we did not have to explicitly specify every possible number of occurrences of \( e_4 \) (which is infinite to begin with). The same can be done for cases like \( S_1 \) where the number of repetitions is fixed. For example by writing \( S_1 = e_1 e_2 e_3 (e_4 | e_4 e_4) \). Such a notation prevents the programmer from explicitly listing the repeated events and instead simply specify the number of repetitions. Additionally, the framework is able to use an efficient implementation in such cases, for example by using a loop counter at runtime to remember the number of occurred repetitions instead of creating and concatenating the same automaton several times.

In terms of our [DSL] a block statement declared with a positive integer \( n \) describes the language formed by concatenating \( n \) instances of the language specified by the block body (after any specified constraints and requirements have been applied). This forms the operation of a repeat block. For example, if the programmer expects an execution of the form \((e_1 e_2 e_3 e_4 e_5)\), the repeating sequence may only be declared once as “repeat 3 body expect e_1 e_2 e_3”, thus declaring the language formed by concatenating 3 instances of the language of the block body \{e_1 e_2 e_3\}.

### Constructing an Automaton for a Repeat Block

Suppose that a [DFA] \( M_B \) recognizes the language \( B \) of a block body after any header statements of the block have been applied. What we do know about the actual language \( S \) described by the entire block with the repeat operation applied is that it is a language formed by concatenating \( n \) instances of \( B \). Hence an execution in \( S \) is formed by taking any \( n \) executions in \( B \) and concatenating them. Consequently, a [DFA] \( M_S \) recognizing \( L(S) \) can be constructed by concatenating \( n \) copies of \( M_B \).

Consider as an example the specification \( S = \text{“repeat 2 body expect } e_1 e_2 \text{end”} \). \( L(S) = \{e_1 e_2 e_1 e_2\} \). The language described by the block \( B \) before the repeat operation is invoked is \( L(B) = \{e_1 e_2\} \). However the sequence is expected twice, resulting in the final language recognized by the automaton \( M_B M_B \) where \( M_B \) is an automaton that recognizes \( L(B) \).

### 3.4.3 The Kleene Closure on Executions

Using the repeat statement without an integer creates the second block type that invokes the Kleene closure on the block body’s language. The framework thus matches zero or more occurrences of executions that belong to the described
language $S$ (that is the language of the block body after header constraints have been applied). Analogous to the Kleene closure on a language, we denote such a statement $S^*$ and define its language $L(S^*)$.

For example, given a block $B$ with $L(B) = \{e_1, e_2\}$, $L(B^*)$ describes the set consisting of all execution containing only the events $e_1$ and $e_2$ — i.e. $\{\varepsilon, e_1, e_2, e_1e_2, e_2e_1, \ldots\}$. Since the closure on a language matches zero or more occurrences, the language $L(B^*)$ always includes an empty execution $\varepsilon$ (containing no events) regardless of $L(B)$.

The addition of the closure operation into our test specification necessarily introduces nondeterminism when performing tests. Consider the closure $L(S^*)$ on some block $S$. An equivalent automaton $M_{S^*}$ at runtime that matches some initial execution $E_0 \in L(S)$ must transition to a next state that implies the current state of the CUT without having access to future events which might be yet to occur. In such a case, the automaton must correctly guess between two options — a transition to a final state signalling that it is done matching executions in $L(S)$, or a transition to a next state that expects to match another execution $E_1 \in L(S)$.

**Constructing an Automaton for a Kleene Block**

In accordance with the construction of a finite automaton for the Kleene closure on a language [21], we build a $\varepsilon$-NFA as $M_{S^*}$ to recognize the closure on an execution set described by some block $S$. We start with the automaton $M_S$ that recognizes $L(S)$ and transform $M_S$ into $M_{S^*}$ by introducing two types of $\varepsilon$-transitions corresponding to the automaton’s choices. We add an $\varepsilon$-transition from the start state $q_0$ of $M_S$ to every final state of $M_S$ allowing the automaton to go directly to the final state when it guesses that all executions have been verified. We also form $\varepsilon$-transitions from each final state $q_n$ back to $q_0$, allowing any number of execution sequences in $L(S)$ to be verified by the automaton.

As an example, figure 3.5 shows an $\varepsilon$-NFA $M_{S^*}$ recognizing the language described by the specification “repeat body expect $e_1e_2$”. The automaton transitions directly from $q_0$ to $q_2$ if no more executions occur while the path from $q_0$ to $q_2$ via $q_1$ is traversed $n$ times where $n$ is the number of consecutive executions of the form $e_1e_2$ that are observed at runtime. Such an automaton could be directly converted to a DFA or simulated directly by keeping a set of current states that the automaton could possibly be in as facilitated by the eclosure mechanism [21].

**3.4.4 Unordered Executions**

Accurately predicting the order of events that make up an execution becomes problematic in message-passing systems comprised of multiple components due
to the inherent asynchrony involved. This is especially problematic in asynchronous distributed environments where there are no upper bounds on computation and message transmission time \cite{45, 28}. However, this problem becomes more manageable in the scope of unit testing since the only observed events are local to a single component in the system — the CUT. Inevitably, the class of verifiable scenarios by the framework are limited to those of local properties of the CUT. For example, scenarios verifying global properties of algorithms, which likely involve assertions on properties across several components, are not specifiable. Nonetheless, it sufficiently serves the purpose of unit testing.

In the scope of unit testing a single component in a message passing system, we consider inherent problems of nondeterminism such as the unpredictable scheduling of components, lack of upper bounds on computation steps and message delays, but only as they pertain to the events observed locally at the CUT’s interface. For example, a set of outgoing requests from the CUT to a set of peer components may expect a set of incoming responses. The order in which the requests are sent or the responses arrive at the CUT may not be accurately specifiable when writing the test case. Hence a need to explicitly specify a set of unordered events as an execution for nondeterministic scenarios.

<table>
<thead>
<tr>
<th>Event → $e_i$</th>
<th>unordered $e_i^+$ end</th>
</tr>
</thead>
</table>

Listing 3.5: Specifying unordered set of events.

Listing 3.5 generates an unordered sequence of events as specified between the \texttt{unordered} and a matching \texttt{end} keyword. Since the order of events do not matter, the language described by the unordered statement consists of all permutations of the originally specified sequence. As an example, the following statement \texttt{"unordered e$_1$ e$_2$ end"} describes the language \{e$_3$e$_4$, e$_4$e$_3$\}. Therefore the statement \texttt{"expect e$_1$ e$_2$ unordered e$_3$ e$_4$ end e$_5$"} describes the language \{e$_1$e$_2$e$_3$e$_4$e$_5$, e$_1$e$_2$e$_4$e$_3$e$_5$\}.

**Constructing an Automaton for Unordered Executions**

Consider the statement $S =$ \texttt{"unordered e$_1$ e$_2$ end"} with $L(S) = \{e_1 e_2, e_2 e_1\}$. Figure 3.6 shows an automaton $M_S$ recognizing an equivalent language. Generally, the statement \texttt{"unordered e$_1$ e$_2$... e$_n$ end"} for $n$ events $e_1$ to $e_n$ describes a language containing $n!$ executions — each a permutation of the originally specified sequence. As a result, a \textbf{DFA} $M$ describing an equivalent language must accept exactly all $n!$ possible executions.
Figure 3.6: An automaton for “unordered $e_1,e_2$ end”. key: \{e_1,e_2\}.

One way to build $M$ is to construct a path for each possible sequence from the start to the end states of the automaton. Each path contains exactly $n$ transitions and $n + 1$ states and each state within a path is used to remember which events have occurred and which are pending. Thus, such a state can be thought of as being associated with a bit string of length $n$ representing the set of specified events $e_1$ to $e_n$ such that the $i$th bit is 1 if event $e_i$ has occurred and 0 otherwise.

Consider the start state $q_0$ of $M$. No events have occurred at this state so it’s bit string has all bits set to 0 — that is the bit string associated with this state is 000...0. Now, suppose at runtime that event $e_n$ occurs first. Then the automaton transitions to the next state $q_δ$ associated with the bit string 000...1 with only the $n$th bit set to 1. A transition from $q_δ$ on event $e_1$ leads to the next state $q_γ = 100...1$ and so on with the final state of the automaton $q_φ = "111...1"$ signalling that all specified events have been matched.

This technique generates a total of $n!$ states where $n$ is the number of specified events. This makes it impractical for even modest values of $n$. In implementation however, the need for extra states is easily avoided since unlike DFAs, the task of remembering the matched events can be accomplished programmatically by the framework. Figure 3.7 shows a more practical scheme using a similar approach to an Extended Finite State Machine (EFSM) [3]. The self-transition labelled $α$ represents the set of events specified by unordered statement. The automaton uses exactly 2 states regardless of the size of $n$ and enables the transition from the start state to the end state only when the trigger condition (all bits in the string have been set to 1) has been met.
3.4.5 Specifying Constraints and Requirements on Executions

<table>
<thead>
<tr>
<th>Hdr</th>
<th>allow $e_i^+$</th>
<th>disallow $e_i^+$</th>
<th>drop $e_i^+$</th>
<th>blockExpect $e_i^+$</th>
</tr>
</thead>
</table>

Listing 3.6: Declaring constraints and requirements on blocks.

Listing 3.6 shows productions for generating statements that appear in the header of a block. The semantics of these statements assume that it is not possible or desirable to predict an exact instance in the block or entire test execution where the constraints must be satisfied. As a result, they apply to an entire block and any nested blocks and can only be specified within the block’s header.

Constraints on Blocks

Within a block some events may be disallowed by the programmer. The occurrence of such events at any point in the block’s execution is undesirable so that the test case should fail. In some other cases, the occurrence of certain events in an execution is not necessary to validate the test case. In fact such events may not even be observed in multiple executions of the same test case. In other words, these events are not required for a successful test case but are allowed if they occur. Finally, the messages associated with some events may be dropped on occurrence — these messages if outgoing from the CUT should not be forwarded to recipients and if incoming, should not be delivered to the CUT. This is particularly useful when writing test logic that verify edge cases and error conditions as it allows the programmer guide the CUT into a vulnerable state.

As is typical of block-structured languages, constraints are only valid within the scope of the block where it was declared as well as its nested blocks — that is a constraint only affects the language described by the block. However, a constraint $C_1$ on an event $e$ in block $B_1$ can be shadowed by redeclaring a new constraint $C_2$ on $e$ in a nested block $B_2$. This means that $C_2$ is valid if $e$ is present within the sub-execution described by block $B_2$ and $C_1$ is valid if $e$ is observed within a sub-execution of $B_1$ but not $B_2$. Conflicts occur when
constraints on the same events are declared within the same block. In such cases, only the last declared constraint is enforced.

As much as statements within block headers describe the execution sequence, they also control the behavior of the framework. For example the disallow statement may be interpreted as an instruction to the framework to fail the test case if any of the specified events occur within the block’s sequence while allow and drop instruct the framework whether or not to forward the messages of the specified events if they do appear in the blocks sequence. In other words, events specified using these constraint statements do not necessarily drive the automaton closer to a final state.

**Constructing an Automaton with Block Headers** Consider two specifications $S = \text{"repeat } 1 \text{ body expect } e_1, e_2 \text{end}\"$ and $C = \text{"repeat } 1 \text{ allow } e_3, e_4 \text{ drop } e_5 \text{ body expect } e_1, e_2\"$ having an equivalent block body to $S$. The automaton $M_S$ shown in figure 3.8 recognizes $L(S)$. To construct the automaton $M_C$ recognizing $L(C)$ as shown in figure 3.9 we incorporate allow and drop statements by adding a self transition, on the set of specified events, to each state of the automaton that represents a statement of the entire block body including nested blocks — in this example these are the states ($q_0, q_1$). In the case of $M_C$, this implies that states recognizing events $e_1$ and $e_2$ are annotated with a self transition on events ($e_3, e_4, e_5$) as specified by the block header. No distinction is made between allow and drop statements in the automaton — the actual difference is in the behavior of the framework implementation at runtime (whether or not it forwards messages).

Since disallow constraints on events cause the test case to fail on occurrence, an equivalent transition from the automaton’s perspective would be labelled with the constrained events and lead to an error state from the current state. Such transitions are implicit as mentioned in section 2.4.1. Just as with allow and disallow constraints, these transitions would be included on each state within the block.

**Requirements on Blocks**

The blockExpect statement shown in listing 3.6 is used to specify nondeterministic scenarios. Consider that I have an expected execution $E = (e_1, e_2)$ of my component and additionally, I expect some event $e_0$ to occur somewhere within this execution. Lets call $E$ a deterministic execution and $e_0$ a nondeterministic event (comprising a nondeterministic execution containing a single event). I may
Figure 3.9: Extending Figure 3.8 with headers “allow e3 e4 drop e5”.

specify the final expected execution to be \((e_0 e_1 e_2)\|(e_1 e_0 e_2)\|(e_1 e_2 e_0)\), capturing all possible positions that the event \(e_0\) can occur relative to the ordered events in the execution. Unfortunately, specifying such executions using this technique becomes infeasible for even modest numbers of nondeterministic events due to the large number of possible permutations of the final execution.

Instead of explicitly trying to list all the possible executions, one can group the deterministic execution \(E_D\) as a single block and using the blockExpect statement, list the nondeterministic events, comprising a nondeterministic execution \(E_N\) for that block. This places a requirement on the framework to successfully exit the block when all events in \(E_D\) have occurred in their correct order and all events in \(E_N\) have occurred in any order. As a result, the previous example can be specified as “repeat blockExpect \(e_0\) body expect \(e_1 e_2\) end” with \(E_D = \{e_1 e_2\}\) and \(E_N = \{e_0\}\) — blockexpect specifies \(E_N\) while \(E_D\) is specified in the block body. Since the events in \(E_N\) are not specified in any relative order to the deterministic event within the block \(E_D\), this statement can be used to describe a wider range of non-deterministic scenarios compared to using unordered statements.

Constructing an Automaton with Block Requirements: Consider the specification \(S = \textnormal{“repeat } 1 \textnormal{ blockExpect } e_3 e_4 \textnormal{ body expect } e_1 e_2 \textnormal{ end”}\). \(L(E_N) = \{e_3 e_4, e_4 e_3\}\) and \(L(E_D) = \{e_1 e_2\}\). Let \(n\) be the number of nondeterministic events as specified by blockExpect and \(d\) be the number of deterministic events specified in the block body — here \(n = d = 2\).

To incorporate blockExpect statements into an automaton, note that any execution \(e_n \in L(E_N)\) can interleave with an execution \(e_d \in L(E_D)\). \(L(S) = \{s \mid s\text{ is a permutation of “}e_1 e_2 e_3 e_4\text{” and } e_1\text{ appears before } e_2\}\). An automaton \(M\) recognizing \(L(S)\) must thus recognize only the sequences containing exactly the events \(e_1\) through \(e_4\) such that the order imposed in the deterministic events of the block body is preserved.

We start with the automaton \(M_D\) recognizing \(L(E_D)\). Each state in \(M_D\) remembers the next expected (deterministic) event as specified in the block body. To incorporate \(E_N\), note that for each state in \(M_D\), the final automaton \(M\) must remember what nondeterministic events have previously occurred. This can be done using a bit string of length \(n\) where the \(i^{th}\) bit represents whether or not the \(i^{th}\) nondeterministic event has occurred. Thus each state in \(M\) is annotated with a bit string as well as a next expected deterministic event —
leading to $2^n$ states for each deterministic event $e$ as each state for $e$ remembers a unique possible configuration of the bit string.

To define the transition function $\delta$ for $M$, consider each state as a pair $(e, s)$ where $e$ is the next expected deterministic event and $s$ is the bit string associated with that state. Valid transitions from a state $q = \{e, s\}$ are labelled on $e_i$ and any unverified nondeterministic event $e_n$ — that is an event that has not yet occurred. For an event $e$ that occurs at the state $q = \{e, s\}$, if $e = e_i$, then the next state is the state $q_\gamma = \{e_i+1, s\}$ where $e_i+1$ is the next expected deterministic event after $e_i$ or $\phi$ if all deterministic events have now occurred, else if $e$ is the $k^{th}$ nondeterministic event and the $k^{th}$ bit in $s$ is 0, then the next state is $q_\gamma = \{e, s_\gamma\}$ where $s_\gamma$ is a copy of $s$ with the $k^{th}$ bit set to 1. The final state $q_\phi = \{\phi, s_\phi\}$ has all bits in $s_\phi$ set to 1 and $e$ set to the empty event since no events are expected.

Figure 3.10 shows an automaton constructed using this technique for the specification “repeat 1 blockExpect $e_3$ body expect $e_1$ end”; key \{next deterministic event, bitstring\}.

![Figure 3.10: DFA for “repeat 1 blockExpect $e_3$ body expect $e_1$ end”; key \{next deterministic event, bitstring\}.](image)

For specifications having form similar to the example, the technique creates $(d + 1)2^n$ states for the constructed automaton, making it impractical for most cases. Just as with the automaton construction in section 3.4.4, an implementation similar to an EFSM can accomplish the task of remembering previously occurred events using $d + 2$ states regardless of the size of $n$. Such an automaton uses $d+1$ states to represent the body of the specification while the logic for remembering nondeterministic events are delegated to the framework as opposed to the automaton. As a result, we include an additional sink state before the final state that transitions to the final state only when all nondeterministic events have occurred. Figure 3.11 shows such an automaton for our example with a transition from the sink state $q_2$ to the final state $q_3$ being enabled only when all nondeterministic events specified by blockExpect $(e_3)$ have occurred.
3.5 Dependencies

So far we have discussed variants of operations from the field of language-recognition that enable a programmer with the help of a test framework implementation, describe and match expected executions of a component. Realistically however, some form of active participation in the test execution process is required to generate the events that make up the expected execution sequence. Consider the previous examples of the form "If I send a message $m_1$ to the component ...", leading to executions of the form $E = (in(m_1), ...)$. We have not mentioned anything regarding the source of the stimuli event $in(m_1)$ and so far have assumed that such events wondrously occur at the desired time which clearly can not be the case. Consequently, we must find ways for generating such events at test runtime.

3.5.1 Testing with External Dependencies

Normally, the CUT communicates with other peer components at runtime in order to perform its computations. These peers become dependencies when testing and must be present or simulated in order to correctly exercise the CUT. There are several reasons for decoupling the CUT from its dependencies when unit testing however. Tests become less complex to write and maintain when its logic does not additionally need to coordinate peer components in order to reproduce the necessary conditions for performing the test case. Also, testing with such dependencies may cause computations to cross the boundaries of a unit test, for example when a peer happens to be a component on a different network and these network messages are not desirable in the tests. More importantly is the uncertainty that the source of a test failure does not originate external to the CUT — a failure in some dependency may cause the test case to inaccurately verify the CUT.

3.5.2 Component Mocking

When such situations are undesirable, a solution is usually to replace these dependencies with mocks — a deterministic and less complex version of a dependency, offering the minimal functionality of the dependency that facilitate the test execution. This requires no new constructs with respect to our proposed
framework as no distinction is made between a mock and a real dependency in
the system — both parties are simply components interacting with the CUT and
the programmer is free to decide whether or not to mock on a per-dependency
basis. It may also be simpler to forego the mock and use the actual dependency
in some cases after all.

3.5.3 Behavioral Mocking

Instead of mocking the entire dependency as explained in the previous section
3.5.1 a programmer may desire to mock only very specific behavior of the de-
dpendency by inserting messages at specific positions in the execution at runtime.
This is done using the trigger statement as shown in listing 3.7 that instructs the
framework to send the specified sequence of messages $m_i$ to the CUT. In fact,
this operation needs not be limited to the CUT. Specified messages may be des-
tined for any peer component in the test environment as long as the framework
implementation is able to provide some mechanism for identifying the recipient.

This approach is a form of interaction and requires the use of online recog-
nition as discussed in section 3.1. In terms of our automaton, we can then
annotate it with special active states that simply perform some operation upon
entry (in this case send a message) instead of (or in addition to) expecting an
event to allow a transition.

As an example, the statement “expect $e_1$ trigger $m_2,m_3$ expect $e_4$” causes the test
framework to send the messages $m_2$ and $m_3$ in between matching events $e_1$ and
$e_4$. The automaton for this statement is shown in figure 3.12. Active states
have their labels underlined as in the case of state $q_1$.

![Figure 3.12: An automaton incorporating an active state $q_1$ that sends messages $m_2$ and $m_3$ upon entry and then transitions to next state $q_2$.]
3.6 White Box Testing

So far we have only discussed black box testing of components where we verify their correctness based on their behavior (external events in their execution sequence). However, we can also perform white box testing by inspecting the \textit{CUT}'s state at any point in their execution. That is, given a component’s execution \(e_1s_1e_2s_2...e_ns_n\) where \(s_i\) is the internal state of the component after performing event \(e_i\), the programmer can \textit{inspect} any state \(s_i\) — the production for which is shown in listing 3.8.

```
Body \rightarrow \texttt{inspect } \alpha
```

Listing 3.8: Inspecting internal state.

The programmer only needs to specify the desired state(s) to be inspected by placing inspect statements in the specification. Conceptually, each such statement provides an assertion function \(\alpha : S_c \rightarrow B\) taking the \textit{CUT}'s (\(c\)) current internal state \(s_i \in S_c\), such that \(\alpha(s_i) = \text{true}\) iff \(s_i\) conforms to the test specification. However, it may be more convenient to simply pass \(c\) itself to \(\alpha\) and reuse imperative unit testing tools, as appropriate to the programming language in use, performing assertions on \(c\)'s internals to decide whether or not \(c\) conforms to the test specification.

In terms of constructing an automaton, as interactive operations, inspections are incorporated using \textit{active} states mentioned in section 3.5.3 that perform the inspection upon entry to the state and transition to the next state on a successful inspection otherwise the test case fails.

3.6.1 Scheduling Inspections

White box testing using \textit{inspect} and online recognition requires the framework implementation to pause the \textit{CUT}'s execution at runtime while its internal state is being inspected to prevent multiple access to the component’s state. In view of messaging components being scheduled nondeterministically, the framework is also required to enforce the happened-before order \cite{PS} between all preceding events (both internal and external) and the current inspect operation. As components of message-passing systems are usually associated with some form of a message queue, messages associated with previous events of the \textit{CUT} may still be queued at the time of inspection. Inspecting the \textit{CUT} under such an incomplete state would result in an inaccurate verification. For example if a programmer chooses to inspect the internal state \(s_2\) in the execution \(e_1s_1e_2s_2e_3...\), then the framework must make sure that both events \(e_1\) and \(e_2\) have occurred and completed while the actual inspection must take place and complete before event \(e_3\) can commence. The framework implementation must thus provide a
transparent mechanism for ensuring that handlers for all events that happen-
before the inspection at runtime have been executed and completed by the CUT
before performing the inspection.

3.7 Matching Events

We have discussed how our [DSL] can be used to describe execution sequences
over our alphabet of events, facilitating a test framework implementation in ver-
fying a component’s behavior. Also mentioned is the idea that the constructed
automaton transitions to a next state when an event \( e_i \) that occurs at runtime
matches the label \( e_j \) of the traversed transition. Obviously \( e_j \) represents some
event provided in the test specification but how exactly would we match two
events at runtime when simulating our automaton?

In regular expressions for strings over a character alphabet \( \Sigma \), matching two
characters would mean to simply compare both characters to check that they
do have the same bit representation according to the character encoding in use.
We can thus say that given an alphabet \( \Sigma \), an equivalence relation \( \sim_\Sigma \) on
\( \Sigma \) is provided such that \( a \sim_\Sigma b \) iff \( a \) matches \( b \) where \( a,b \in \Sigma \). Thus in our
example of regular expressions using characters, \( \sim_\Sigma = \{(a,b) \mid a,b \text{ have the}
same bit representation}\).

Now in our case of event alphabet \( \Sigma \), each event \( e \) is made up of a direction
\( d \in D \) and an associated message \( m \in M \) where \( D = \{in,out\} \) — here we
will use the notation for an event \( e = (d,m) \). An equivalence relation is thus
defined to be \( \sim_\Sigma = \{(d_1,m_1),(d_2,m_2) \mid d_1 \sim_D d_2 \text{ and } m_1 \sim_M m_2\} \).
Clearly the equivalence relation for \( D \) should be defined as \( \sim_D = \{(in,in),(out,out)\} \)
with constants as \( in \) and \( out \) but defining \( \sim_M \) is not as straightforward since
the equality on messages likely depend on their structure as dictated by the
programming language in use. For example, in Erlang [7], \( \sim_M \) could be imple-
mented using pattern matching of messages while in Java, one would need to
write the intended equals method for each message class or provide an instance
of java.util.Comparator or equivalent function.

Regardless of the method for defining \( \sim_M \), it is likely to have its limitations
in practice. Consider scenarios where we might be expecting only messages that
fulfil some criteria or that can be matched by examining specific parts of their
contents. For example if we only expect a message of a certain class or messages
from a specific sender where the sender field may be examined for a match. Using
a single equivalence relation \( \sim_M \), the programmer would be required to specify
the entire message, including irrelevant fields, to be matched. This is not very
efficient or desirable. Usually, string regular expression implementations pro-
vide predefined matches for common character classes like digit and whitespace
characters to avoid this inconvenience — e.g writing \( \backslash d \) instead of \((0|1|2|\ldots|9)\) to
match a single digit. However, this approach is not directly applicable to our event symbols due to the wide variety of possible messages.

Instead we take a more general approach, allowing the programmer to provide a predicate which we call a matching function $\sigma : M \rightarrow B$ as opposed to an actual message $m_i$. Thus, the programmer specifies $(d_i, \sigma_i)$ and an observed event $e_j = (d_j, m_j)$ is matched iff $d_i \sim_D d_j$ and $\sigma_i(m_j) = true$. As an example, figure 3.13 shows an automaton for the specification “repeat 2 expect (in, \sigma_i) end” where $\sigma_i(m)$ returns true iff the incoming message $m$ was sent by some predefined sender $x$. A successful test case matches exactly two such messages.

3.8 Beyond Regular Tests

So far we are able to describe a wide variety of common scenarios by specifying regular tests with executions such as $E = (out(m_1), in(m_2))$. But messages are data carrying objects, so consider a variant of $E$ where message $m_2$ is a response to $m_1$ and both have to agree on some particular value, say of a field id, that is randomly generated at runtime. The test case should only pass sequences such that the id field of $m_1$ matches that of $m_2$. Since the value of id is generated at runtime, we have no way of statically writing a matching function for $m_1$ in our specification. Such a scenario is typical of a request-response pattern where an outgoing message $out(m_1)$, called the request, is expected to be answered by an incoming response $in(m_i)$.

As a first attempt to capture such scenarios, we provide a function $\mu : M^2 \rightarrow B$ that takes two messages $m_i$ and $m_j$ and returns true iff $m_i$ matches $m_j$. Thus $\mu$ is a matching function that handles the extraction of the id fields, comparing their values for equivalence. At runtime we can place each request on top of a stack when they are observed. Upon receiving a response $rs$ we pop a request $rq$ from the stack, passing both $rq$ and $rs$ to our matching function $\mu$. If both messages match each other then the state transition is successful. This approach is that of a pushdown automaton which was mentioned in section 2.3.5 and as discussed, recognizes context-free languages. In fact the CFG for this attempt is shown in listing 3.9.
Listing 3.9: Specifying request responses in a LIFO manner.

This approach using a stack allows us to capture scenarios of the form $S_1 = \text{“Every request } rq_i \text{ must be immediately answered by receiving a matching response } rs_i\text{”}$ and $S_2 = \text{“Every request } rq_i \text{ must be answered by receiving a matching response in a LIFO manner”}$. Thus we can correctly specify executions such as $E = (\text{out}(m_1)\text{in}(m_1)\text{out}(m_2)\text{in}(m_2)\ldots)$ for scenarios of the form in $S_1$ and $E = (\text{out}(m_1)\text{out}(m_2)\text{in}(m_2)\text{in}(m_1))$ for scenarios in $S_2$.

Unfortunately, these scenarios are still quite restrictive and not particularly realistic. What we would like is to capture arbitrary interleavings of responses for requests such as $S = \text{“Any request } rq_i \text{ must be answered by receiving a matching response } rs_i\text{”}$. This would capture more common scenarios like $E = (\text{out}(m_1)\text{out}(m_2)\text{in}(m_1)\text{in}(m_2)\ldots)$.

The relationship between requests and their responses is referred to as cross-serial dependency in linguistics [49] and it has been shown that such languages are not context-free but can be generated by Context-Sensitive Grammars (CSGs) and accepted by a Linear Bounded Automaton (LBA) [25] [24]. However instead of generating a LBA for such scenarios, the task of remembering pending requests and matching them with incoming responses may be executed programmatically by the framework implementation.

One technique that can be used to accomplish this involves the framework keeping a set $R_q$ of pending requests and upon receiving a response $rs$, removing a request $rq \in R_q$ that matches $rs$ according to the matching function $\mu$. Implementations based on this technique can differ in two main aspects:

**Interleaving with regular executions**: An implementation may choose to explicitly distinguish a request-response execution from regular executions. This means that the programmer must specify some start and end checkpoints within the expected execution (denoted ‘<’ and ‘>’ respectively) between which request messages and their responses must occur. Since request messages are not specifiable, outgoing messages are automatically treated as requests while incoming messages are treated as responses between these checkpoints. For example $E = a < rq_1 rq_2 > b$ say that $rq_1$ and $rq_2$ are request events which must occur after $a$ while their associated responses must occur ($R_q$ must be empty) before event $b$ occurs. Since regular events can not be interleaved with request/response events, this technique is less flexible in nondeterministic scenarios.

Another technique that improves on this flexibility issue may do away with the explicit checkpoints, instead implementing them implicitly with ‘<’ being the start of the execution and ‘>’ being an implicit barrier such as the end of a repeat block or the end of the test case by which $R_q$
must be empty. Until the barrier is crossed in such an implementation, there is no explicit way to identify response messages so on receiving an event $e = (m, in)$ such that $\mu(rq_i, m) = \text{true}$, then $rq_i$ is removed from $R_q$, otherwise $e$ is treated as a regular event and used to simulate the automaton.

**Response behavior:** Being unable to provide matching functions for response messages (similar to requests) at specification time means that 1. response events can not be explicitly identified without consulting $\mu$. 2. responses behaviors of components can not be mocked using the technique described in section 3.5.3 — a response message can not be injected into an execution.

For the former, we mentioned in the previous point that we could treat each incoming message as a response when using explicit checkpoints while we had to check using $\mu$ when using implicit checkpoints. For the latter, an implementation may choose to offer a mechanism for mocking responses by allowing the programmer to provide a mapper function $\omega : R_q \rightarrow R_s$ that generates a response $rs \in R_s$ for a given request message $rq \in R_q$ where $R_s$ is the set of all possible response messages. The generated response $rs$ may then be triggered on a specified port as if it were sent by the mocked dependency — thus mocking that particular behavior of it.

As illustrated in section 4.5, our current implementation of the request-response pattern uses explicit checkpoints while allowing the programmer to mock response behavior.

### 3.9 Ambiguous Test Specifications

The introduction of interactions (sections 3.5.3, 3.6) into our DSL while useful for testing, come with caveats when specifying test cases with inherent non-determinism. Consider a Kleene block $K = \text{"repeat trigger } m_1 \text{ body end"}$ and a conditional statement where both branches begin with interactive operations $C_1 = \text{"either trigger } m_1 \text{ or trigger } m_2 \text{ end"}$. The generated automata for $K$ and $C_1$ are shown in figures 3.14 and 3.15 respectively.

![Figure 3.14: Generated automaton for ambiguous specification “repeat trigger m1, body end”](image-url)
In the first case $K$, the block does not define any clear entry condition as seen in automaton 3.14 — the first state does not consume any input and performs only interactive actions. So we have no reliable means of guessing how many times the transition labelled $m_1$ should be traversed — that is how many times the message $m_1$ should be sent. The second case $C_1$ does not provide any means to select between the either and or branches since both begin with interactions and thus do not have clear entry conditions. Selection of a conditional path to traverse at runtime depends on the event observed at the given state but none is expected in this case (state $q_{A\cup B}$).

![Generated automaton for ambiguous statement “either trigger $m_1$ or trigger $m_2$ end”.](image)

Figure 3.15: Generated automaton for ambiguous statement “either trigger $m_1$ or trigger $m_2$ end”.

Now consider the statement $C_2 = “either trigger $m_1$ or expect $e_1$ end” and the constructed automaton in figure 3.16. It does not have a clear entry condition for the either branch but it does have one for the or branch. At runtime, if the event $e_1$ occurs then the transition labelled $e_1$ is traversed to final state $q_{B1}$, otherwise the message $m_1$ is sent and the automaton follows that transition instead to end up in final state $q_{A1}$. Generally, if a matching event occurs, the automaton follows that transition else it performs an interaction instead (note that if there are multiple interactions at the current state, then we have the same issue as in $K$).

The programmer’s intent can not be inferred by the framework from statements such as those in $K$ and $C_1$ and in fact, the intent in $K$ is questionable to begin with. Specifications like these are deemed ambiguous and the behavior of a framework implementation is left undefined in such cases. Statement $C_2$ on the other hand is not ambiguous due to the clear entry condition of the or branch.
3.9.1 Using Matching Functions

Section 3.7 introduced matching functions into our DSL as a means for matching events by categories. While this approach is very powerful, it is accompanied with the possibility of writing ambiguous specifications. Specifically, a programmer must ensure that any automaton created from the specification does not contain a state with two transitions labelled with matching functions \((d, \sigma_1)\) and \((d, \sigma_2)\) respectively such that there exists a message \(m\) with \(\sigma_1(m) = \text{true} = \sigma_2(m)\).

To illustrate using a naive example, consider the specification \(M = \text{"either trigger } m_1 \text{ or expect } e_1 \text{ end"} \) such that \(\sigma_1(m) = \text{true}\) iff message \(m\) was sent by some sender \(x\) and \(\sigma_2(m) = \text{true}\) iff \(m\) has type \(\textit{\#class}\) assuming we are using Java. The generated automaton is shown in figure 3.17. Say at start state \(q_{A \cup B}\), an incoming message \(m\) of type \(\textit{\#class}\) and sent by \(x\) is observed. Then the automaton is rendered nondeterministic. Note that although we may choose to simulate the automaton as an NFA with the next states being \(\{q_{A1}, q_{B1}\}\), this is likely not what the programmer had in mind. More importantly, we cannot convert such an automaton to a DFA since it is not possible to know the entire set of messages for which the specified matching functions return true. Consequently, specifications of this form are declared ambiguous as well.
3.10 Terminating Test Cases

In section 3.1, we alluded to the difficulty in having the automaton correctly decide when a test case should terminate in online testing. Consider an automaton recognizing the sequence \( S = e_1 e_2 e_3 e_4 e_4^* \). Observing the sequence \( R = e_1 e_2 e_3 e_4 \) at runtime puts the automaton in an accept state but there might be more events \( e_i \) yet to occur such that if \( e_i \) matches \( e_4 \) then the automaton should transition to another accept state otherwise the test case should fail.

We can see that we cannot terminate and declare the test case a success on recognizing \( R \) as this would mean that we infact have an automaton recognizing \( R \) and not \( S \), nor would we want to risk possibly waiting indefinitely for \( e_i \) to occur before making a decision. Had we performed offline testing instead, the end of the execution sequence would simply be marked with an end of stream event \( \$ \) so that only once this event is observed can the automaton safely declare the test case a success if it currently is in an accept state. Note that this problem can also be present in any state that expects some event \( e_i \) to occur before following a transition since the automaton may also have to wait indefinitely for any event at all to occur.

A practical solution to this problem would be to define a sufficiently large timeout value, most likely specifiable by the programmer on a per-testcase basis, such that the automaton can safely make a decision if no events occur within that time period.
Chapter 4

KompicsTesting - An Implementation

In order to evaluate our approach we have implemented a prototype, called KompicsTesting, for the Kompics component framework. Specifically, KompicsTesting is implemented for the reference implementation of Kompics [4] which is written in Java. However, there are other implementations of Kompics in Python and Scala [27] which are not considered in this thesis and we simply refer to the reference implementation as Kompics throughout this chapter.

We start with a description of the event symbols that make up our alphabet of executions, followed by the architectural and design decisions that enable our prototype enforce the operational semantics of the approach presented so far. Finally we show some examples of using KompicsTesting’s DSL implementation according to the CFG in listing 3.1 for creating actual test specifications.

4.1 Ports and Directions

Here we make the notion of events concrete for our implementation. First we relegate the concept of events in the Kompics model to messages in order to avoid any ambiguity when referring to actual events as described so far by our approach. Ports are an essential part of events in Kompics but our theoretical model provides no notion of them. We must thus incorporate them into our alphabet. For a given test case, let $M$ be the set containing all possible, observable messages, $P$ the set of ports declared by all peer components including the CUT and a set of directions $D = \{ \text{in, out} \}$.

As mentioned in section 2.5.1, Kompics provides a type system for events in the system so we restrict our alphabet to the relation $\Sigma$ on $M \times P \times D$ consisting of triples $(m \in M, p \in P, d \in D)$ where $p$ is declared by the CUT and the port
type of $p$ allows messages of type $m$ in direction $d$. Thus $(m, p, in) \in \Sigma$ iff some component $c$ provides $p$ and $p$ allows $m$ in the negative direction, or $c$ requires $p$ and $p$ allows $m$ in the positive direction. KompicsTesting is also implemented in Java so $D$ is implemented as Java enums while messages and ports are simply provided as references in a specification.

4.2 Runtime

4.2.1 The Proxy Component

In our approach as presented so far in this thesis, it is assumed that messages going in and out of the CUT are not only observable by an implementation but also interceptable for cases like allow and drop since recipients of such messages may not be allowed to receive them. KompicsTesting uses a special component called the proxy, as the main component 2.5.4 responsible for intercepting and observing messages to and from the CUT. The mechanism used to achieve this is facilitated by Kompics’ component heirarchy. As the first component to be created by the Kompics runtime, the proxy acts as a composite component in charge of creating and bootstrapping the CUT and its dependencies as children components.

4.2.2 Internal Port Implementation in Kompics

As a bidirectional interface, what looks like two sides of a declared port $\alpha = (+, -)\] in Kompics is actually internally represented by a pair of port instances linked to each other. As a result, from here on we shall refer to each such port instance simply as a port. For each linked port pair $(+, -)$ (technically port instances) one is owned by the component $C$ that declares $\alpha$ while the other is owned by the parent component $P$ of $C$. If $\alpha$ is provided by $C$, the inside port $-\alpha$ is owned by $C$ while the outside $+\alpha$ port is owned by $P$ and conversely, if $\alpha$ is required by $C$, the inside port $+\alpha$ is owned by $C$ while the outside port $-\alpha$ is owned by $P$. On receiving a message on a port, the port’s owner is scheduled for execution.

To illustrate, we reuse the example in section 2.5.1 and the pingpong port declaration in 2.7. Consider the proxy component with a child component pinger that requires the port pingpong. Component pinger, the owner of the positive port ‘+’ subscribes its handlers and triggers messages on that port while its parent proxy may subscribe its own handlers to the negative ‘-’ port and trigger messages on it. A message triggered on one port is immediately passed to the port’s pair and any connected channels to that pair, executing any handlers subscribed to the pair. Passing a message through a channel simply means to trigger that message on the destination port of that channel. Thus triggering a pong message on the ‘-’ port causes the handler handlePong of pinger to be executed and triggering a ping message on the ‘+’ port causes handlePing of proxy to be executed.
4.2.3 Intercepting Messages

We may now be able to see from the previous example that establishing a parent-child relationship between the proxy and all other components enables a sufficient degree of control over the messages that go in and out of a child. The proxy can subscribe its own handlers to every child’s ports and listen for messages that pass through them. Coupled with the fact that creating channels between any two ports in the test environment must be done through the test framework, the proxy is able to completely isolate the CUT from its dependencies, intercept messages at its interfaces and only forward those which have not been explicitly disabled by the specification.

Let us assume that our CUT is the pinger component and it has a dependency component ponger. So ping messages are outgoing from the CUT and pong messages are incoming to it (from ponger). We would like handlers intcpPing and intcpPong to intercept these respective types of messages at test runtime.

As a first attempt in intercepting pinger’s messages we connect the outside ports of both components via a channel c as would be done in a production environment. We could subscribe our intercepting handlers intcpPing and intcpPong on the outside ‘-’ and inside ‘+’ port of pinger respectively. As a result, Kompics executes our handlers whenever a message is received on either port which is exactly what we want in terms of being able to listen for messages. Unfortunately, our intercepting handlers will always be executed along with those subscribed by intended recipient and we will not be able to optionally forward messages — the recipient would already have received or be about to receive the message by the time our handlers execute.

Our solution is to connect the proxy directly to any dependencies, leaving the CUT with no direct connections. Due to Kompics’ channel broadcast mechanism, peers are unaware of the source of received messages and recipients of triggered messages. Thus by mirroring all ports declared by the CUT on the proxy, dependencies may be directly connected to the proxy while the CUT remains isolated. Handler intcpPing and generally handlers intercepting outgoing messages keep track of the connected channels and component, manually forwarding messages to dependencies as requested by the specification. Handler intcpPong and generally handlers intercepting incoming messages are subscribed to the proxy’s mirrored inside ports instead and relays received messages to the CUT as requested by the specification.

4.2.4 Scheduling

Some scheduling decisions must be made in order to enforce the semantics of our DSL. In section 4.2.3 we described the process by which messages are intercepted but not how we determine whether or not an intercepted message should be forwarded.
bd.repeat(5)
    .allow(m1, p1, in) // block header
    .body() // block body
    .expect(m2, p1, in)
    .expect(m3, p2, out)
    .end();

Listing 4.1: A test specification in Java created using a builder pattern.

KompicsTesting runs the generated automaton on a single control thread while all other components except for proxy are scheduled on a thread pool. Component proxy uses a calling thread scheduler — meaning that its handlers are executed immediately on the same thread of the component triggering a message on any of its ports. An event queue keeps track of a partial execution sequence at any instance at runtime — that is, it contains those events that have occurred but have not yet been consumed by the automaton. The job of the proxy’s intercepting handlers become simply to place an observed event on the queue. Since these components are scheduled independent of the automaton, the automaton can progress at its own pace, consuming events from the queue only when this is required to transition to some next state. Essentially, intercepting handlers place events in the queue while the automaton removes them from the queue. Only when an event is consumed from a queue is it actually handled — that is optionally forwarded to the intended recipient(s) as determined by the current state’s transitions.

This same set up also allows KompicsTesting to enforce the happened-before order of events as explained in section 3.6.1 as it ensures that a state inspection predicate is executed at the right point in the sequence of events without risking concurrent access or unexpected interleavings.

4.3 Specification Builder

As our target Kompics implementation is written in Java, we had to embed our DSL in Java even though it does not provide optimal platform for building DSL. The approach we took to implementing the DSL is the builder pattern [17]. It includes the full CFG as described in section 3.2 with additional constructs like connecting and creating components. Listing 4.1 gives an example of how the builder pattern is used in Java to create an actual specification. Here the described execution is $S = (e_1 e_2 e_1 e_3 e_1)^5$ where $e_1 = \text{in}(p_1, m_1), e_2 = \text{in}(p_1, m_2)$ and $e_3 = \text{out}(p_2, m_3)$. Here, $p_i$ are port instances while $m_i$ are event messages. Later in section 4.6 we show how matching functions may be used instead of explicitly match an equivalent message.

4.4 Creating and Executing a Test Case

In KompicsTesting, the first step of verifying a component is by creating an instance of the TestContext class by providing a component definition of the CUT.
Listing 4.2: Creating and Executing a Test Case

Listing 4.2 illustrates the process of creating and running a test case using the example in section 2.5.1. Here the CUT is a ponger component and line 1 creates the previously mentioned test context for it by providing its component definition class Ponger while line 2 retrieves a reference to the created ponger instance. Line 3 creates a pinger component as a dependency and line 4 connects their PingPongPort ports. After all such setup activities have been carried out, methods for describing the execution of the CUT may be then called.

We assume that every ping and pong instance is associated with an id integer field and that we require our ponger to respond to every ping request having a non-negative id, with a pong message having the same id. Otherwise the response may either be a pong message with id 1 followed by another with id 2, or any number (including zero) of pong messages with id 3. We assume that the equals method of the message classes have been overridden accordingly.

Now, for this particular test case we verify both cases where a ping request has a negative and non-negative id. Line 8 expects a ping with id 0 to be sent by the connected pinger dependency on start and line 9 expects a response with the same id. To provoke the second type of behavior, we inject a negative ping message in line 10 and use the conditional statement (lines 11 – 18) to specify both possible cases with line 12 and 13 matching the two pong messages with ids 1 and 2 as required and lines 15 – 17 use the repeat method without any arguments to create a kleene block that matches a single pong message of id 3 any number of times.

If we assign each expected event letters a through d as shown using comments, then the described language of this test case is $L = ab(cde^*)$ with the interaction ping message being positioned after event b. Finally calling the check method as shown in line 19 runs the test case, returning true iff the observed execution is a member of the language $L$ described by the test case.
4.5 Answering Requests

KompicsTesting provides specialized support for the request-response pattern described in section 3.8. Here we instead allow the response messages to be generated by the programmer and triggered on some specified destination port on receiving a matched request — the response messages thus mock the behavior of some other component. The programmer provides a mapper function $\rho : Rq \rightarrow Rs$ for each expected request that returns a unique response if provided a message that is matched otherwise it returns $null$ — $\rho$ thus acts both as the matching function $\mu$ and the mapper function $\omega$ in section 3.8. It also uses explicit checkpoints with start and end points marked by calling methods `expectWithMapper` and `end` respectively and for each expected request in the pattern, a request class, source and destination port and mapper function is provided. Listing 4.3 demonstrates basic usages of this pattern.

4.6 Applications

To evaluate the practicality of our prototype, we presented it to one of the main developers of Kompics for use in testing a data streaming application [40]. In this section, we present one of the implemented test cases, highlighting the following recurring sources of errors when developing components in Kompics and how they are mitigated by KompicsTesting early in the testing process:

**Memory Leaks** For long-running programs like messaging components, it is important that unused memory is discarded for reuse to prevent performance degradation over time.

**Unsubscribed Handlers** It is very common for programmers to develop a handler and subsequently forget to explicitly subscribe it to the intended port (or sometimes subscribe it to the wrong port). These types of mistakes are usually very difficult and annoying to debug using traditional testing tools.

**Untriggered Events** Somewhat related to the previous point is omitting program logic to trigger an event or even worse, triggering an event at the wrong time or port.

4.6.1 The Kompics FSM

In Kompics, fundamental actions of components such as handling of messages are detached from the actual protocol in which the component participates. This design decision makes it somewhat difficult to understand interactions in complex protocols involving large amount of messages. However, Kompics provides a Finite State Machine (FSM) abstraction [41] that can be used to model protocols in a more visual and easy to understand manner. This abstraction becomes
especially valuable since we can map the states of the protocol FSM to those in the automaton generated for our test case helping us better understand our illustration. A FSM can thus be seen as an Application Programming Interface (API) for the actual underlying protocol and as a visual aid for understanding the protocol and for writing tests.

In Kompics, multiple instances of the same FSM can run within a single component and a set of handlers for a single event can be provided, of which only a subset may be subscribed depending on the current state of an instance. It supports protocol internal state (PIS) which is protocol state shared between all the states of a single FSM instance and protocol external state (PES) state shared across multiple FSM instances running within the same component. Since handlers of a single component are executed sequentially in Kompics, the user is guaranteed that only a single handler across all FSM instances of a component is run at a certain time, preventing concurrent access to the PIS and PES. An executed handler thus decides the state transitions based on the received event and the current state (PIS or PES).

4.6.2 A Streaming Application

The tested system is a data streaming application [41] written in Kompics — an application that can transfer data between two machines. The entire application is split across four components: A Storage component ST takes care of writing data to the user’s storage device, Network component NT in charge of transferring data over the network, Transfer Manager component TM in charge of managing the lifecycle of a data transfer and a User component User representing an external user that is able to interact with the rest of the application through TM.

The Transfer Manager Component

The CUT for this testcase is TM as it is the most interesting component, interacting with all other components as shown in figure 4.1. Component User sends requests to TM to start and stop a transfer while TM in turn sends requests to ST and NT to setup/clean storage and network transfers respectively. Thus from the perspective of TM, the following interactions with other components are possible:
The expected interaction is as followed: When a User wants to start a data transfer, it sends message u1 to TM and expects a message u2 when TM starts downloading the requested data. Once the download completes, the application starts uploading the data to peers (think BitTorrent) and User expects message u3. At any point within this lifecycle of a transfer, User can send u4 to cancel the transfer.

Internally, TM on receiving the initial transfer request u1, sets up storage by sending st1 and waiting for confirmation st2 and then sets up the actual network transfer by sending nt1 and waiting nt2. Eventually the (download) transfer completes and it receives nt3 from NT after which it sends u3 to User. On receiving a cancel request u4 from User, TM performs clean up of shared external state by first sending nt4 and waiting for nt5, then sending st3 and waiting for st4. Figure 4.2 visualizes the protocol using Kompics’ FSM implementation. For brevity we have only specified the most important transitions and states.

The expected behavior of TM in this test case is $S = TC$ where $T = \langle in(u_1) out(st_1) in(st_2) out(nt_1) in(nt_2) out(u_2) in(nt_3) out(u_3) \rangle$ and $C = \langle in(u_4) out(nt_4) in(nt_5) out(st_3) in(st_4) \rangle$. So T is an execution sequence for a complete download transfer while C is an execution sequence for a cancel scenario. $S$ becomes a completed transfer followed by a cancel scenario. Listing 4.3 shows the actual specification written in KompicsTesting. Component User is an external dependency connected in line 1 while the specific behavior of components ST and NT are mocked. Since the request id of message u1 is generated at runtime, line 3 expects the event in(u1) by specifying a matching function any(u1.class) that matches a message of the request class. After each interaction an inspection of the instance state is performed (lines 4, 8, 12, 16, 18, 22, 26). Two-way interactions with mocked components ST and NT are performed using the request-response pattern described in section 4.5. For example line 5 - 7 expects message st1 and provides a mapping
Listing 4.3: Use case test specification

function storage(ST1.class) from the received message to a $st_2$ message which is then triggered on the same port $pB$ — thus mocking a response from $sr$. In line 14, the behavior of $nt$ is also mocked by sending the download complete message $nt_3$ to the CUT after which message $u_3$ signalling download complete is expected (line 15) and the cancel scenario commences in line 19. The PES is also verified to not have any memory leaks by running this scenario 100 times (line 2) and having the PES inspected after each transfer (line 26).

A large class of errors detected by KompicsTesting stem from illegal sequences of messages — where no message is received on time or the wrong message appears at some position in the expected sequence. These are usually caused by 1) an unexpected message being triggered, 2) an expected message is omitted or 3) the expected message is delayed in which case the developer may provide a custom timeout value if the initial estimation was insufficient.

As mentioned previously, these errors are difficult to find by simply inspecting code or using traditional tools but KompicsTesting can easily detect an observed event different from what is currently expected or that observing no events at all imply nonconformance to the test specification and consequently fail the test case. We also showed how the test case incorporated state inspections after every interaction and repeatedly executed the scenario, aiding us in detecting any lurking memory leaks. The mistake of forgetting to subscribe handlers is as easily detected from the observed sequence of messages and state inspections as an omission to execute a handler would be visible any of these two avenues.
Figure 4.2: Visualizing a data transfer lifecycle.
Chapter 5

Conclusions and Future Work

In this thesis project we have presented a platform independent approach to unit testing message-passing systems. Unlike previous techniques such as model checking and state-space exploration, our presented approach allows the programmer to specify the particular state-space of the messaging-component to be covered. Thus problems like state-explosion which may make verification intractable are avoided at the expense of being unable to guarantee full test coverage.

We have developed a DSL similar to regular expressions to facilitate the specification of test cases that include interactions with the component under test and the test environment, black-box testing and inspections of internal state to support white-box testing. Using methods from formal language theory, we have described ways to generate a testing automaton from a test specification described in our DSL.

To evaluate our approach we have developed a prototype implementation in Java, targeting the Kompics component framework. Here, we also described considerations in applying the theoretical approach to practical framework implementations. Employing an example use case, we showed the practical application of our approach to a real world problem and how common errors found in typical scenarios could be mitigated early in the development process.

5.1 Future Work

Testing Multiple Components

Since the scope of this project was limited to the verification of a single component even though multiple cooperating components may be present at the
test runtime, the next step would be to generalise the approach presented even further by allowing simultaneous tests of multiple messaging-components. More importantly, such an improvement should allow the specification of properties that cross the boundaries of a single component (and by extension a single testing automaton). For example, this is a necessary part of being able to correctly verify distributed behaviors such as an agreement among a set of nodes running the same protocol.

**Performing Combinatorial Tests**

Interleaving of events is a very common occurrence in message-passing systems and unfortunately just as common, a cause of errors in these systems. Component implementations must be fully aware of such possibilities and perform correctly in their presence and we would like to be easily able to verify this. One approach is to perform combinatorial tests by allowing the programmer mark sub-sequences of events that are eligible for interleaving and any associated constraints with regards to the allowed order between any two marked events. The framework then verifies that a component under test conforms to the specification under every legal permutation of the marked sub-sequences. This would likely involve the use of constraint programming techniques to avoid the combinatorial explosion of possible sequences needed for such a verification.

**Dealing with Failures**

The presented approach also does not explicitly deal with partial failure (and its simulation) of components in the testing environment — an unfortunately common phenomenon in any message-passing system. To easily verify components in the presence of failures, we would like to extend our approach with a scheme to easily simulate various failure modes of components during test execution. This is for example helpful when explicitly verifying the behavior of a component under test in the presence of failures.

5.1.1 Implementation improvements

The prototype implementation presented in this project only targets a particular implementation of the Kompics component framework. Testing frameworks based on the presented approach could be implemented in other implementations of Kompics and more importantly in different message-passing models and platforms. For example creating framework implementations on Akka and Erlang on the actor model. These systems can further be compared with each other and the findings used to improve the approach.

5.2 Benefits, Ethics and Sustainability

These days, an improvement in the overall quality of software systems is demanded by companies and consumers. In this thesis project, we have shown
how the quality of even the most complex message-passing systems can be verified independently at the granularity of components — a technique that is very effective in detecting errors and consequently improving the quality of the system and decreasing the development costs. Therefore testing frameworks developed based on the approach presented in this thesis could be used to improve the process of finding and fixing errors at early phases in the software development process.

Further development of our approach and by extension, frameworks based on it, may facilitate

• more effective developers and testers as using these tools will lead to a less error-prone style of developing software systems.

• efficient development processes as the detection of errors will be done as soon as they are introduced into the system. Coupled with new agile methods, these tools will provide the user with rapid feedback on the correctness of software system and allow errors to be swiftly fixed.

No ethical issues were encountered with respect to the development and result of this project as the project solely deals with the techniques and the implementation of testing tools.
Bibliography


