Master thesis
Computer Science

Radboud University

Coalgebra Testing

Author:
Joris den Elzen
s4481038

First supervisor/assessor:
dr. J. Rot
jrot@cs.ru.nl

Second assessor:
prof. dr. B.P.F. Jacobs
b.jacobs@cs.ru.nl

October 2, 2022

Abstract

This thesis aims to generalize methods for black-box testing of state machines, pioneered by Moore in 1954, to arbitrary coalgebraic systems. For this purpose we use a dual adjunction between coalgebraic structures and modal logics, where formulas are used as tests, and together with a one-step semantics of the logic we acquire dual semantic maps between coalgebra and logic. We propose a testing method that is complete for systems with bounded size, and demonstrate it on two types of coalgebra; Mealy machines and labelled transition systems.

Contents

Chapter 1

Introduction

A significant part of software development consists of testing a system and trying to convince oneself that the system behaves 'as it should.' Usually, testing consists of giving a system a varied set of inputs and seeing whether it then produces the expected outputs. The first formalisation of this problem was by Moore [19] in 1956 and he called it *conformance checking*. We start by describing the exact behaviour of a system using a model in the form of a state machine, and perform various inputs to check if our implementation conforms to the specified model. The implementation is treated as a black-box; we assume it to be a state machine, but we can only observe the outputs for a given input. Moore's methods have been optimised many times over, for an overview see [16].

As testing theory is a well-developed field in computer science, it provides frameworks for different types of systems, such as nondeterministic state machines [20]. However, as much of the work is aimed at real-world application, a general abstraction for these different types of systems is lacking. Chapter 2 of [18] provides a generalisation of many testing frameworks for Mealy machines, and provides one of the starting points for this thesis. The aim of this thesis is then to generalise further, towards a framework that gives an abstract notion of a complete test suite for any system that can be described by a *coalgebra* [21].

Coalgebras are a categorical notion that can be viewed as an abstraction of state based systems [9]. Where tests for finite state machines are sequences of inputs, a natural way to specify properties of coalgebras is through coalgebraic modal logic, through a dual adjunction between coalgebras and algebras that describe modal syntax [7, 12]. After defining the semantics for a single 'layer' of modal operators, called the one-step semantics, the adjunction then provides semantic relations between coalgebras and their corresponding logic. This duality has been applied in [4] to develop an algorithm that constructs a model coalgebra from a black-box, a generalisation of the L^* algorithm by Dana Angluin [2], which constructs

a deterministic finite automata from a system viewed as a black-box. We would now like to do the same for conformance testing (also called finite state machine testing, which we will be referred to as FSM testing) methods, such as the ones described in [18] and [20]. Our goal is to develop a generalised method of testing coalgebraic systems, using duality for test semantics.

In our setting, we assume that we have full access to the successors of the current state of the system under test. In the case of Mealy machines, this just means that for every input we get access to the state that this input transitions to. For labelled transition systems, this allows us to access all the states that are reached by a transition with a certain label. This assumption is comparable to the 'complete testing assumption' [17] in nondeterministic FSM testing [20].

Chapter 2 introduces FSM testing with a method for testing Mealy machines, simple state-machines where transitions are labelled with input and output. We then discuss some preliminary category theory and introduce the notion of coalgebras. The last two sections discuss the two most important tools we use to construct our algorithm; the logical adjunction and the construction of the base [6], which is used to iterate through the black-box system to access its states for testing.

Our main contribution is chapter 3, where we develop our coalgebraic testing method. We start by defining the necessary parts of the framework, provide an algorithm that 'runs' the test suite, and conclude with an example test run of two very different state-based systems; the Mealy machines that you will have seen in chapter 2, and nondeterministic finitely-branching labelled transition systems. We will see that the logics used for testing are very different; for Mealy machines with input alphabet I and output alphabet O, we use an O-valued logic where formulas are sequences of symbols from I, and their truth values correspond to the last output symbol for that sequence of inputs. For labelled transition system with labels A, we use a variation of Hennessy-Milner logic [5] with negation and conjunction, where modal operators are labelled by symbols from A.

Chapter 2

Preliminaries

2.1 Automata Testing

Since coalgebras are abstractions of state-based systems, we first summarise how testing of deterministic input-output automata called Mealy machines is done. Testing, in essence, is gathering information from a black box. You might want to find out through testing in what state the machine is, check whether a hypothetical model of the machine is correct, or even build such a model through tests (this last problem is known as "model learning").

In our setting, by testing we mean *conformance* testing. Given a desired specification of a system in the form of a finite state machine, testing should allow us to say with certainty whether an implemented system, of which we can only observe its behaviour, *conforms* to the given specification. This is also called FSM-testing or *black-box* testing, as the implementation is treated as a black-box; we cannot observe its inner workings directly. The specification FSM is also called a *model* of the system.

Moore [19] discusses some early testing problems and gives us a complete test suite (a set that, given a model, determines whether a black-box is equivalent to this model) for Mealy machines. Further refinement by Chow and Vasilevski (independently) reduced the exponential size of the suite to polynomial order [24, 8]. We will use this method, called the W-method, as our starting point. The second chapter of Joshua Moerman's thesis [18] on FSM-based test methods provides a uniform generalisation of testing methods for Mealy machines, which can be considered an appropriate place to start when developing a generalised method of coalgebra testing. As such, this section briefly summarises the framework discussed there, giving a definition of a complete test suite for Mealy machines and proving sufficient conditions for completeness.

Finally, we will illustrate the test method with an example test suite for a given specification Mealy machine. We will return to this example after developing our generalised testing framework for coalgebras, illustrating the

similarities between the concrete method and the generalised method.

2.1.1 Mealy Machines

Mealy machines are systems that consist of states, where transitions between states take an input from input alphabet I and produce an output from set O. In this way, Mealy machines describe functions on sequences of input characters, hence why they are also called sequential machines. The set of sequences (sometimes also called words) of symbols from I is denoted by I^*. Concatenation of two sequences v and w is denoted as vw.

Definition 2.1. A *Mealy machine* is a tuple $(S, \delta, \lambda, s_0)$, where S is the set of states, $\delta : S \times I \to S$ is the transition function, and $\lambda : S \times I \to O$ the output function. The state $s_0 \in S$ is the *initial state*.

Using these we can define functions on sequences of input, where ϵ denotes the empty word:

$$\begin{aligned} \delta^* &: S \times I^* \to S \\ \delta^*(s, \epsilon) &= s \\ \delta^*(s, aw) &= \delta^*(\delta(s, a), w) \\ \lambda^* &: S \times I^* \to O* \\ \lambda^*(s, \epsilon) &= \epsilon \\ \lambda^*(s, aw) &= \lambda(s, a)\lambda^*(\delta(s, a), w) \end{aligned}$$

with $a \in I$ and $w \in I^*$.

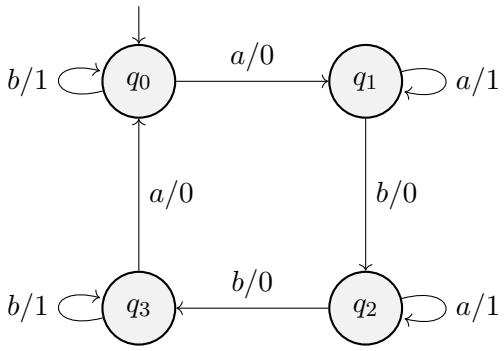


Figure 2.1: An example of a Mealy machine.

Figure 2.1 shows an example of a Mealy machine, with input alphabet $I = \{a, b\}$ and output alphabet $O = \{0, 1\}$. A transition labelled with $a/0$ means that the machine takes a as an input and outputs 0. Table 2.1 shows the transition and output functions with their output for each input.

State	Input	δ	λ
q_0	a	q_1	0
q_0	b	q_0	1
q_1	a	q_1	1
q_1	b	q_2	0
q_2	a	q_2	1
q_2	b	q_3	0
q_3	a	q_0	0
q_3	b	q_3	1

Table 2.1: Transition function $\delta : S \times I \to S$ and output function $\lambda : S \times I \to O$ of the Mealy machine in figure 2.1.

2.1.2 Testing Mealy machines

Let $M = (S, \delta, \lambda, s_0)$ and $M' = (X, \delta_X, \gamma_X, x_0)$ be two Mealy machines. Running a test $t \in I^*$ then amounts to checking whether $\lambda^*(s_0, t) = \lambda^*_X(x_0, t)$. The function λ^* essentially defines the *behaviour* of a state. We write $s \sim x$ if for all $w \in I^*$ we have $\lambda^*(s, w) = \lambda^*_X(x, w)$. We call this *behavioural equivalence*. Two Mealy machines are equivalent if their initial states are equivalent; $s_0 \sim x_0$. For a set of inputs W we say s and x are *behaviourally equivalent w.r.t.* W if they agree on all $w \in W$. We denote this by $s \sim_W x$. M is *minimal* w.r.t. behavioural equivalence if for every pair of states s_1 and s_2 in S we have that $s_1 \sim s_2$ implies that $s_1 = s_2$. That is, every state in M behaves uniquely w.r.t. to the other states; there are no redundant states in M.

We call a collection of tests $t \in I^*$ a *test suite*:

Definition 2.2. A *test suite* for a Mealy Machine M with input alphabet I is just a set T of input sequences, $T \subseteq I^*$.

We will assume that our specification has n inequivalent states, and that the black-box system under test has at most $m = n + k$ states. The desired property of our test suite is m-completeness: that for any black-box with a size bounded by m that is inequivalent to our model, there is a test in the suite that provides a counterexample to equivalence.

Definition 2.3. A test-suite T is *m-complete* for model $M = (S, \delta, \lambda, s_0)$ if for any inequivalent $M' = (X, \delta_X, \gamma_X, x_0)$ with at most m states, there is a test $t \in T$ such that $\lambda^*(s_0, t) \neq \lambda^*_X(x_0, t)$.

We can also concatenate sets of sequences; for sets P and Q this is denoted as $P \cdot Q$, and defined as $P \cdot Q = \{vw \mid v \in P, w \in Q\}$. The test

suite we will discuss consists of a number of parts. The first part is the *state cover*:

Definition 2.4. A *state cover* for a Mealy machine $M = (S, \delta, \lambda, s_0)$ is a set of input sequences $P \subseteq I^*$ such that for all states $s \in S$ that are reachable from s_0, there exists a sequence $p \in P$ such that $\delta^*(s_0, p) = s$.

The state cover is basically a set that contains inputs that can configure the machine to be in any of its states, assuming they are reachable from the initial state. Note that the smallest state cover contains exactly one sequence for every reachable state, and a state cover can always be constructed by taking $I^{\leq n}$, where n is the number of states of the Mealy machine. For the example in figure 2.1, a state cover is $P = \{\epsilon, a, ab, abb\}$.

The *transition cover* Q can be constructed as $Q = P \cdot I = \{pa \mid p \in P, a \in I\}$. This set of inputs traverses all transitions of the model. In the case of the Mealy machine in figure 2.1, the transition cover is $\{a, baba, abb, abba, abbb\}$.

When we are testing some black-box machine M', we want the state reached by some $p \in P$ in M' to correspond to the state reached by p in M, and similarly for Q, where we want the transitions in both machines to line up. This is where the second part of the test suite comes in:

Definition 2.5. A *characterisation* of a Mealy machine $M = (S, \delta, \lambda, s_0)$ is a set of inputs $W \subseteq I^*$ such that for all inequivalent pairs of states s_1 and s_2 in S there is a word $w \in W$ that *separates* s_1 and s_2: $\lambda^*(s_1, w) \neq \lambda^*(s_2, w)$.

A characterisation is used to distinguish between inequivalent states in an implementation. A characterisation for the example in figure 2.1 would be $W = \{ab, ba\}$, since $\lambda^*(q_0)(ab) = 00$, where $\lambda^*(q_1)(ab) = \lambda^*(q_2)(ab) = 10$ and $\lambda^*(q_3)(ab) = 01$, so ab already separates q_0 and q_3, and separates both q_0 and q_3 from q_1 and q_2. The sequence ba then separates q_1 and q_2; $\lambda^*(q_1)(ba) = 01$ and $\lambda^*(q_2)(ba) = 00$.

We will use a *bisimulation* to prove the completeness of the test suite.

Definition 2.6. Let M be a Mealy machine. A relation $R \subseteq S \times S$ on states S of M is called a *bisimulation* if for every $(s, t) \in R$ we have:

- equal outputs: $\lambda(s, a) = \lambda(t, a)$ for all $a \in I$.
- related successor states: $(\delta(s, a), \delta(t, a)) \in R$ for all $a \in I$.

For an example of a bisimulation, see the machines in figures 2.1 and 2.2. An example of a bisimulation between these two machines is:

$$R = \{(q_0, p_0), (q_1, p_1), (q_1, p_4), (q_2, p_2), (q_3, p_3)\}$$

The following lemma shows that we can use bisimulation as a tool to prove behavioural equivalence.

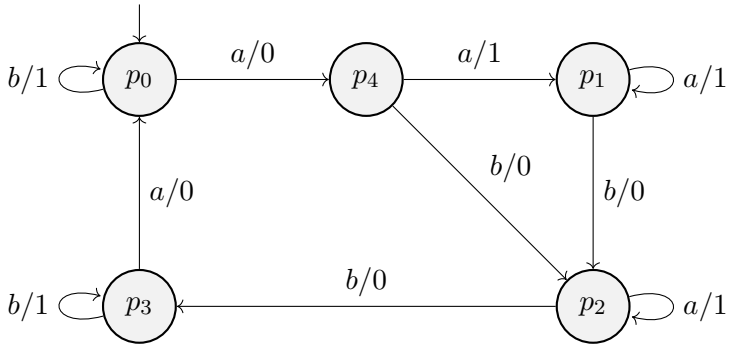


Figure 2.2: A Mealy machine behaviourally equivalent to the machine in figure 2.1.

Lemma 2.7. *Let R be a bisimulation. If $(s,t) \in R$ then $s \sim t$.*

Proof. Let R be a bisimulation. We want to prove $\forall w \in I^*.\forall (s,t) \in R.\lambda^*(s,w) = \lambda^*(t,w)$. We do this by induction on w. The base case, the empty word ϵ, is easy: $\lambda^*(s,\epsilon) = \epsilon = \lambda^*(t,\epsilon)$. We now want to prove that $\forall aw \in I^*.\forall (s,t) \in R.\lambda(s,aw) = \lambda(t,aw)$. Let $aw \in I^*$, $(s,t) \in R$. Then we have $\lambda(s,a) = \lambda(t,a)$ by the equal output property of R. We also have $(\delta(s,a), \delta(t,a)) \in R$ by the related successor states property of R. By the induction hypothesis we get $\lambda(\delta(s,a),w) = \lambda(\delta(t,a),w)$, giving us $\lambda(s,a)\lambda(\delta(s,a),w) = \lambda(t,a)\lambda(\delta(t,a),w)$. □

We will now construct the test suite. Let

- $M = (S, \delta, \lambda, s_0)$ be our model
- P be a state cover for M
- W be a characterisation of M
- $Q = P \cdot I = \{pa \mid p \in P, a \in I\}$ be the transition cover
- $I^{\leq k} = \{w \in I^* \mid |w| \leq k\}$ be a set sequences of inputs from I with length k or smaller.

We start by first concatenating the $I^{\leq k}$ to the state cover P. This allows us to reach all possible extra states in the implementation machine (as we assume it to have at most k states). We then concatenate W to this set, in order to do equivalence testing between states reached by $P \cdot I^{\leq k}$. We also do this with Q to check that the equivalent states are reached after every possible transition. The resulting test suite is then:

$$T = (P \cup Q) \cdot I^{\leq k} \cdot W \tag{2.1}$$

What follows is a somewhat simplified version of proposition 31 in [18]. It is simplified in that it only accounts for the case of the W-method, whereas the cited proposition covers a multitude of testing methods.

Proposition 2.8. *Let M be a minimal Mealy machine with state cover P, transition cover $Q = P \circ I = \{pa \mid p \in P, a \in I\}$, and W a characterization of M. Let $T = (P \cup Q) \circ I^{\leq k} \circ W$ be our test suite. For any Mealy machine $M' = (X, \delta_X, \lambda_X, x_0)$ with $m = n + k$ states, if $s_0 \sim_T x_0$ then $s_0 \sim x_0$.*

Proof. We start by proving that $\{\delta_X^*(x_0, p) \mid p \in P \cdot I^{\leq k}\}$ contains all states in M'. Note that for any p and q in P such that $\delta^*(s_0, p) \not\sim_W \delta * (s_0, q)$ we also have that $\delta_X^*(x_0, p) \not\sim_W \delta_X^*(x_0, q)$. Because W is a characterisation of M this happens at least n times, and thus P reaches at least n different states in M'. Concatenating P with $I^{\leq k}$ ensures that we reach the k extra states.

Define $R = \{(\delta^*(s_0, p), \delta_X^*(x_0, p) \mid p \in P \cdot I^{\leq k}\}$. We prove that this is bisimulation. Note that for all (s, x) in R we have that $s \sim_W x$, and that for every x in X there is an s in S such that $(s, x) \in R$, because $P \cdot I^{\leq k}$ reaches all states of M'.

Let $(s, x) \in R$. Then since $s_0 \sim_{P \cdot I^{\leq k+1}} x_0$ (because $P \cdot I^{\leq k+1} = Q \cdot I^{\leq k} \subseteq T$) we get that $\lambda(s, a) = \lambda_X(x, a)$ for all $a \in I$. For the successors, fix $a \in I$ and let $s_2 = \delta(s, a)$ and $x_2 = \delta(x, a)$. As stated before, we know there is a $t \in S$ such that $(t, x_2) \in R$. So we get that $s_2 \sim_W x_2 \sim_W t$. Since W is a characterisation, we get $s_2 \sim t$ and by minimality of M they should be the same state, so $(s_2, x_2) \in R$. □

Completeness of T directly follows: if M' is inequivalent to M, in other words, $s_0 \not\sim x_0$, then by proposition 2.8 we have that $s_0 \not\sim_T x_0$. So there is a test $w \in T$ such that $\lambda^*(s_0, w) \neq \lambda_X^*(x_0, w)$.

2.1.3 Example

For the demonstration of the test suite, our specification Mealy machine will be the one in figure 2.3, taken from [18], with state cover $P = \{\epsilon, a, aa, b, ba\}$.

Before we can start testing, we need a characterisation for this Mealy machine. The sequence 'aa' separates m_0 for all other states, as it is the only state that outputs a 0 after this sequence. In the same way, state m_1 is distinguished from the rest of the states by sequence 'a', m_3 by 'c' and m_4 by the sequence 'ac'. For m_2 there is no single sequence, as sequences starting with b or c have equal outputs for states m_3 and m_4, respectively. When taking an a transition we get the same situation with m_3 and m_4 swapped. However, since we already have sequences for all states that distinguish them from m_2, we know that the set $W = \{aa, a, c, ac\}$ will also distinguish m_2 from the other states.

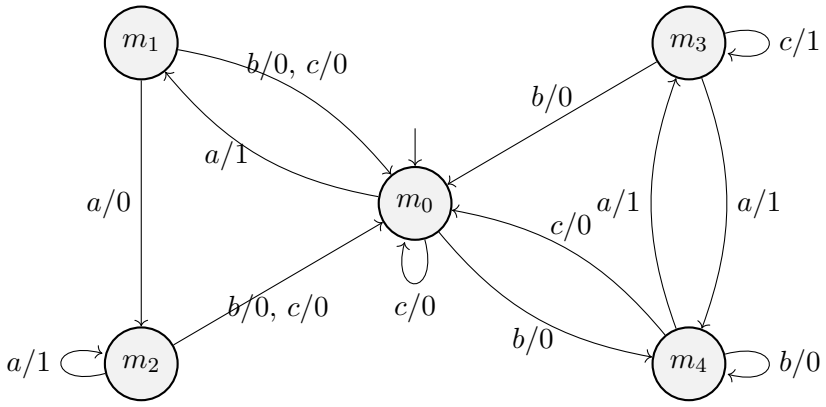


Figure 2.3: Specification Mealy machine

We construct the test suite as shown in the previous section, $T = (P \cup Q) \circ I^{\leq k} \circ W$, with $k = 0$:

$$\begin{aligned} P =& \{\epsilon, a, aa, b, ba\} \\ Q = P \cdot I =& \{a, b, c, aa, ab, ac, aaa, \\ & aab, aac, ba, bb, bc, baa, bab, bac\} \\ T =& \{aaaaa, aaaac, aaac, aabaa, aabac, \\ & aabc, aacaa, aacac, aacc, abaa, abc, acaa, \\ & acac, acc, baaaa, baaac, baac, babaa, babac, \\ & babc, bacaa, bacac, bacc, bbaa, bbac, bbc, bcaa, \\ & bcac, bcc, caa, cac, cc\} \end{aligned}$$

For brevity we have not included any prefixes in T, as it is prefixed closed (since P, Q and W are also prefix-closed). In total T contains 169 sequences. Now lets consider the faulty implementation in figure 2.4. The transition function is the same as the model except at y_3 we have $\delta_Y(y_3)(a) = (1, y_3)$ where it should be $(1, y_4)$. We notice this when we run test $baac$; the output for the model is 0110, while for the implementation we get 0111. We can view this test as containing two parts: the part that reaches the states m_3 and y_3, which is ba, the part that takes the next transition, a, and the part from the characterisation that separates m_3 from m_4, the input sequence c. This illustrates the purpose of all the different components in the test suite.

2.2 Category Theory and Coalgebra

In order to generalise existing testing techniques for state-based systems, we generalise these systems to coalgebras. Coalgebras are a category-theoretic description of state-based systems. As such, in order to understand coalgebras, we will need some category theory. First, we will briefly summarise the

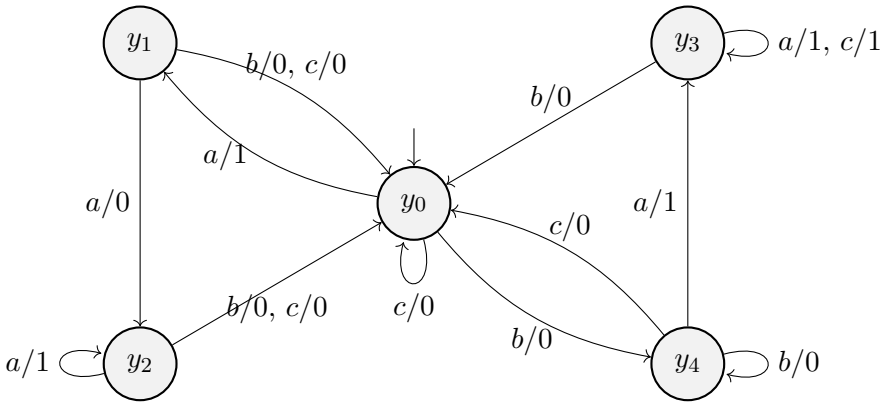


Figure 2.4: Faulty implementation (Y, γ_Y, y_0) of specification (M, γ_M, m_0) (figure 3.1

.

basic definitions of category theory, discussing objects, morphisms, functors and the like. We will introduce some basic categorical tools that we will use in the development of our coalgebraic testing theory, such as subobjects and adjunctions. A great introduction to category theory that was used extensively as reference during this thesis is the book by Awodey [3]. Afterwards we will discuss the method of coalgebra [21, 9].

2.2.1 Categories and Functors

Definition 2.9. A *category* $\mathcal{C}$ consists of:

- *Objects*, often denoted in capital letters A, B, C, etc. The collection of objects of $\mathcal{C}$ is written as $Ob(\mathcal{C})$.
- Arrows between objects called *morphisms*, denoted by lowercase letters f, g, etc. They have a domain $dom(f)$ and a codomain $cod(f)$ which are objects of $\mathcal{C}$. The notation $f : A \to B$ describes a morphism f with domain A and codomain B such that:
  - Each object A has an associated *identity* morphism:
    $$Id_A : A \to A$$
  - For any two morphisms $f : A \to B$ and $g : B \to C$ such that $cod(f) = dom(g)$ there exists a composite morphism:
    $$g \circ f : A \to C$$
  - Composition is *associative*:
    $$f \circ (g \circ h) = (f \circ g) \circ h$$
    for all $f : A \to B$, $g : B \to C$, $h : C \to D$.

– The identity is the *unit* of composition:

$$f \circ Id_A = f = Id_B \circ f$$

for any $f : A \to B$

Morphisms can have various properties, some of which are defined below:

Definition 2.10. Let $\mathcal{C}$ be a category with objects A, B and let $f : A \to B$ be a morphism in $\mathcal{C}$. Then:

- if for every pair of parallel morphisms $g_1, g_2 : Z \to A$ we have that $f \circ g_1 = f \circ g_2$ implies $g_1 = g_2$, we call f a *monomorphism* or a *mono*. We denote this with $f : A \rightarrowtail B$
- if for every pair of parallel morphisms $g_1, g_2 : B \to Z$ we have that $g_1 \circ f = g_2 \circ f$ implies $g_1 = g_2$, we call f an *epimorphism* or an *epi*. We denote this with $f : A \twoheadrightarrow B$.
- if there exist a *inverse* morphism $f^{-1} : B \to A$ such that $f^{-1} \circ f = \mathsf{Id}_A$ and $f \circ f^{-1} = \mathsf{Id}_B$, then f is an *isomorphism* or an *iso*. If there exists an isomorphism between two objects A and B, we write $A \cong B$.

Most of our work will be done in the category Set, where objects are sets and morphisms are functions. These three notions can be viewed as generalizations of injective, surjective and bijective functions in the category Set, respectively.

There are also maps *between* categories, called functors. Since they take a category as an argument and produce a category as a result, they are defined on both objects and morphisms, and they have to preserve identities, composition with associativity and the composition unit.

Definition 2.11. A *functor*

$$F : \mathcal{C} \to \mathcal{D}$$

is a map from objects and morphisms from $\mathcal{C}$ to objects and morphisms from $\mathcal{D}$, such that:

- $F(f : A \to B) = F(f) : F(A) \to F(B)$
- $F(\mathsf{Id}_A) = \mathsf{Id}_{F(A)}$
- $F(g \circ f) = F(g) \circ F(f)$

So in the definition A and B are objects from $\mathcal{C}$, f and g are morphisms from $\mathcal{C}$, $F(A)$ and $F(B)$ are objects in $\mathcal{D}$, and $F(f)$ and $F(g)$ are morphisms in $\mathcal{D}$. It is easy to check that functors themselves also compose associatively,

and that for every category we have an identity functor $\mathsf{Id}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$. This gives rise to the category of categories and functors, Cat.

The identity functor is an example of an *endofunctor*, a functor that stays inside the same category. Another endofunctor that we will use often is $A \times (-)$, the product of an object with a fixed object A. For Set, this is the Cartesian product:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

And on functions $f : X \to Y$:

$$A \times (f : X \to Y) : A \times X \to A \times Y$$
$$(A \times f)(a, x) = (a, f(x))$$

This has the required properties:

$$(A \times Id_X)(a, x) = (a, x) = Id_{A \times X}(a, x)$$
$$(A \times g) \circ (A \times f)(a, x) = A \times g(a, f(x)) = (a, g(f(x))) = A \times (g \circ f)(a, x)$$

Another common functor that we will use often is taking a constant *exponent* $(-)^A$. In Set, for B a set, B^A means the set of all functions from A to B:

$$B^A = \{f : A \to B\}$$

On functions:

$$(f : X \to Y)^A : X^A \to Y^A$$
$$f^A(g)(a) = f(g(a))$$

Here f^A takes a function $g : A \to X$ as an argument, since $g \in X^A$. It results in a function that takes $a \in A$ as an argument and returns $f(g(a)) \in Y$, so a function $A \to Y$, which is in Y^A. Let's check if it preserves identity and composition:

$$(\mathsf{Id}_X)^A(g)(a) = \mathsf{Id}_X(g(a)) = g(a) = \mathsf{Id}_{X^A}(g)(a)$$
$$(g^A \circ f^A)(h)(a) = g^A(f^A(h))(a) = g(f^A(h)(a)) = g(f(h(a))) = (g \circ f)^A(h)(a)$$

Since functors and categories are a category themselves, functors can also be composed, for example if $FX = A \times X$ and $GX = X^B$ then $G \circ FX = (A \times X)^B$.

2.2.2 Coalgebra

Given a endofunctor B on a category $\mathcal{C}$ we define a *B-coalgebra* as:

Definition 2.12. A *B-coalgebra* is an object X of $\mathcal{C}$ together with a morphism $\gamma : X \to BX$. A *pointed* coalgebra is a coalgebra with an *initial state* $x_0 : 1 \rightarrowtail X$.

Coalgebras are a generalisation of state-based systems. The state space is represented by X, and the transition structure is represented by γ. Different types of automata are represented by different functors. For example, deterministic finite automata over an alphabet A are exactly the coalgebras for the functor $B(X) = 2 \times X^A$ in the category of sets, where 2 denotes a set with two distinct elements, final and non-final states. An automaton with an initial state can then be defined as a pointed coalgebra $(X, \gamma : X \to 2 \times X^A, x_0)$. For a state $x \in X$ we then get $\pi_1(\gamma(x)) \in 2$, which states whether x is a final state or not, and $\pi_2(\gamma(x))(a) \in X$, which determines the state that is reached from x by an a-transition.

Mealy machines are described by coalgebras of the functor $B(X) = (O \times X)^I$, where I and O are the input- and output alphabet respectively. A Mealy coalgebra $\gamma : X \to (O \times X)^I$ is then a function that takes a state and for every input gives an output and a new state. In contrast to DFAs, where the states are labelled by 2, here the transitions are labelled by both I and O.

Morphisms in $\mathcal{C}$ that preserve the coalgebraic structure between two coalgebras are called homomorphisms and are defined as such:

Definition 2.13. An B-coalgebra *homomorphism* from coalgebra (X, γ_X) to coalgebra (Y, γ_Y) is a morphism $h : X \to Y$ such that $Bh \circ \gamma_X = \gamma_Y \circ h$, as illustrated in diagram 2.2.

$$\begin{array}{ccc} X & \xrightarrow{h} & Y \\ \downarrow{\scriptstyle \gamma_X} & & \downarrow{\scriptstyle \gamma_Y} \\ BX & \xrightarrow{Bh} & BY \end{array} \tag{2.2}$$

B-coalgebras and their homomorphisms form a category $\mathsf{CoAlg}(B)$. The terminal object of this category is called the *final* B-coalgebra, a B-coalgebra (Z, γ_Z) such that for every B-coalgebra (X, γ_X) there is a *unique* coalgebra homomorphism h from X to Z.

For Mealy machines the final coalgebra is the set of functions that take a nonempty input word and give an output symbol: $\{f \mid f : I^+ \to O\}$. A state in a Mealy machine is entirely defined by the final output it gives on all nonempty input sequences, this is what the final coalgebra represents. We will see that we use this in the logic for testing Mealy machines as well.

We have already seen the notion of bisimilarity for Mealy machines in the previous chapter. Here we generalise it to coalgebras:

Definition 2.14. A *bisimulation* between coalgebras (X, γ_X) and (Y, γ_Y) is a relation $R \subseteq X \times Y$ for which there exists a morphism $g : R \to BR$ such

that projections $\pi_1 : R \to X$ and $\pi_2 : R \to Y$ are coalgebra homomorphisms (as seen in diagram 2.3).

$$\begin{array}{ccccc} X & \xleftarrow[\pi_1]{} & R & \xrightarrow[\pi_2]{} & Y \\ \downarrow{\scriptstyle \gamma_X} & & \vdots{\scriptstyle g} & & \downarrow{\scriptstyle \gamma_Y} \\ BX & \xleftarrow{B\pi_1} & BR & \xrightarrow{B\pi_2} & BY \end{array} \tag{2.3}$$

States of a coalgebra are called *bisimilar* if they are contained in a bisimulation. For pointed coalgebra, the notion of equivalence is bisimilarity of their initial states.

There exists a strong relationship between homomorphisms and bisimulations. The following theorem from [22] illustrates this:

Theorem 2.15. *Let (X, γ_X) and (Y, γ_Y) be two coalgebras. A morphism $f : X \to Y$ is a homomorphism if and only if the graph of f is a bisimulation.*

For us this means that we can prove bisimilarity, in other words behavioural equivalence, between two coalgebras by providing a homomorphism. We will use this notion extensively when proving the correctness of our algorithm.

The dual notion of a F-coalgebra is an *F-algebra*, which is a morphism $\alpha : FX \to X$ with $F : \mathcal{C} \to \mathcal{C}$ an endofunctor and X an object in $\mathcal{C}$. Just like coalgbras, F-algebras form a category $\mathsf{Alg}(F)$. The *initial* object of this category is called the initial algebra for F. We will use algebras to describe the syntax of our logic for testing coalgbras.

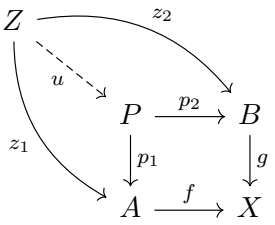
2.2.3 Subobjects

Subobjects can be viewed as a categorical generalisation of the notion of subsets in the category of sets. They are also structure preserving; a subobject in the category of groups is a subgroup. Subobjects are usually defined as equivalence classes of monomorphisms. As such, before we define subobjects themselves, we should first consider this equivalence. For two monomorphisms $s_1 : S_1 \rightarrowtail X$ and $s_2 : S_2 \rightarrowtail X$, we write $s_1 \leq s_2$ if there exists a morphism $u : S_1 \to S_2$ such that $s_1 = s_2 \circ u$. We say that s_1 *factors through* s_2. If this u is an isomorphism, we say that s_1 and s_2 are equivalent, $s_1 \equiv s_2$. The subobjects of X are then defined as the equivalence classes of monomorphisms into X with respect to $\equiv$. As such, the subobjects of an object X form a preorder. In this thesis we will usually use a single representative monomorphism to identify a subobject. The collection of subobjects of an object X is denoted by $\mathrm{Sub}(X)$. An important property of the categories that we are working in for our method is *well-poweredness*. This means that for all objects $X \in \mathcal{C}$ the collection $\mathsf{Sub}(X)$ of subobjects of X is a set.

Pullbacks and pushouts

We define *intersection* of a family of subobjects $\bigwedge_i \{s_i : S_i \rightarrowtail X\}$ by the greatest lower bound with respect to the order $\leq$. In a complete category, these intersections can be computed by a (wide) pullback. A pullback is defined as follows:

Definition 2.16. A *pullback* of morphisms $f : A \to X$ and $g : B \to X$ consists of arrows $p_1 : P \to A$, $p_2 : P \to B$ such that $f \circ p_1 = g \circ p_2$ and for any other set of arrows $z_1 : Z \to A$, $z_2 : Z \to B$ there exists a unique morphism $u : Z \to P$ such that diagram 2.4 commutes.



(2.4)

It is easy to see that monomorphisms are preserved by pullbacks, so if we take f and g to be subobjects of X, $p : P \rightarrowtail X$ (defined by $p = f \circ p_1$) is again a subobject. By the universal property of the pullback, we see that P is then actually the greatest lower bound, or the intersection $f \wedge g$ as discussed in the previous section.

If we consider the structure defined in 2.4 as a *binary* pullback, we can consider *wide* pullbacks to be pullbacks of arbitrary cardinality; given an indexed collection of morphisms $\{s_i : S_i \to X\}_{i \in I}$, the wide pullback is an object P with an indexed collection of projections $\{p_i : P \to S_i\}_{i \in I}$ (called the *limiting cone*) where $s_i \circ p_i$ is independent of i, in other words, for all i, j in I we have $s_i \circ p_i = s_j \circ p_j$, with the universal property that for any other cone $\{q_i : Q \to S_i\}$ we get a morphism $f : Q \to P$ such that for all i we have $p_i \circ f = q_i$, In this way, intersections of arbitrary cardinality can be computed using these wide pullbacks.

In a similar fashion, the pushout as the dual of pullbacks can be used to compute the union/join of an indexed collection $\bigvee_{i \in I} \{s_i : S_i \rightarrowtail X\}$, as it is the (wide) pushout of the set of morphisms $\{o_i : \bigwedge_{i \in I} S_i \to S_i\}$. This works if the category we are working in is *adhesive* [14], which, among other things, means pushouts also preserve monomorphism. Set is an example of an adhesive category, as is any topos [15]. This pushout then has exactly the properties we want the join to have: it induces a mono $\bigvee_{i \in I} s_i : \bigvee_{i \in I} S_i \rightarrowtail X$ such that it is above all s_i in the preorder of subobjects, and any other upper bound of the collection of subobjects is above the join.

Pullbacks and pushouts are examples of *limits* and *colimits*, respectively; given a diagram in $\mathcal{C}$, a limit is an object L in $\mathcal{C}$ with a collection of morphisms to every object in the diagram, such that the diagram with L and all

these morphisms added still commutes, together with a universal property, which means that for any other object K with such a collection of morphisms, there is a unique map from K to L. A colimit is the dual notion of this; just reverse all the morphisms. A category $\mathcal{C}$ is called *complete* if every small diagram $\mathcal{C}$ has a limit. A diagram is small if its component objects and morphisms can be collected in a set.

2.2.4 Adjunctions

Central to our testing theory is an adjunction between the coalgebra functor and a functor that describes modality of a logic. Adjunctions are an important category theoretic notion that generalise many mathematical phenomena. In this section we will give a canonical definition of adjunctions between functors, and later, in chapter 3, we will describe in detail the adjunction between coalgebras and logics.

An intuitive way to understand adjunctions is that they give a 'best approximation' of objects and morphisms of a category in another category. Common adjunctions are between forgetful functors (functors that just throw away some structure, such as going from a monoid to its underlying set) and free functors (for example the functor that generates the free monoid from a set).

Definition 2.17. An *adjunction* consists of two functors $F : \mathcal{C} \to \mathcal{D}$ and $U : \mathcal{D} \to \mathcal{C}$ and an isomorphism

$$\phi : \mathrm{Hom}_{\mathcal{D}}(FC, D) \cong \mathrm{Hom}_{\mathcal{C}}(C, UD) : \psi$$

The *unit* $\eta : \mathsf{Id}_{\mathcal{C}} \to U \circ F$ and *counit* $\epsilon : F \circ U \to \mathsf{Id}_{\mathcal{D}}$ of the adjunction are natural transformations defined by:

$$\eta_C = \phi(\mathsf{Id}_{FC})$$
$$\epsilon_D = \psi(\mathsf{Id}_{UD})$$

Example 2.18. Let $\mathcal{C}$ be a category with binary products. Let $A \in \mathcal{C}$ be a fixed object. We want to find an adjoint of the functor $FX = X \times A$. This means that we need a functor U such that we have a natural bijection between morphisms of the form $X \times A \to Y$ and $X \to UY$. Let's try *currying* A; $UY = Y^A$. Now we can define the bijection by using the universal property of exponent. Let $\epsilon : Y^A \times A \to Y$ be the *evaluation* of the exponent. Then for any arrow $f : X \times A \to Y$ we get a unique arrow $\tilde{f} : X \to Y^A$ such that $\epsilon \circ (\tilde{f} \times 1_A) = f$. This gives rise to the bijection where $\phi(f) = \tilde{f}$. We can define ψ explicitly as $\psi(g) = \epsilon_Y \circ F(g)$.

In the category of sets and functions we can define all this more explicitly. We can just define $\phi(f)(x)(a) = f(x, a)$. It is easy to check that this is a natural bijection.

For our purposes we are interested in a special type of adjunction, the *dual* adjunction or *duality*. A duality is an adjunction between *contravariant* functors, i.e., functors that change the direction of the morphisms. A contravariant functor $F : \mathcal{C} \to \mathcal{D}$ is just a functor $F : \mathcal{C}_{\text{op}} \to \mathcal{D}$. An example is the *contravariant powerset functor* 2^-, which sends a function $f : X \to Y$ to its preimage function $2^f : 2^Y \to 2^X$, where $2^f(S) = \{x \in X \mid f(x) \in S\}$ with $S \subseteq Y$.

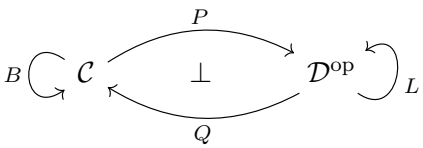
2.3 Coalgebras and Modal Logic

Modal logics [5] are often used for describing properties of transition systems. In our framework the modal logic used could be languages over an alphabet as a logic for deterministic finite automata, or functions that take a finite sequence of input symbols and produces an output symbol for Mealy machines. Modal logics essentially provide the *testing language* of our framework.

2.3.1 Logical Adjunction

We use dual adjunctions to connect coalgebras to their corresponding modal logic [12]. On one side of the adjunction we have the category $\mathcal{C}$ where the coalgebras live, and on the other side of the adjunction we have the category $\mathcal{D}$ where the logics live. The categories are connected by adjoint contravariant functors P and Q, forming adjunction $P : \mathcal{C} \rightleftarrows \mathcal{D}^{op}$. This means that we get a bijection $\phi : \mathcal{C}(X, Q\Delta) \rightleftarrows \mathcal{D}(\Delta, PX) : \psi$ that is natural in both $\mathcal{C}$ and $\mathcal{D}$. The functor P then maps coalgebras from $\mathcal{C}$ to collections of formulas in $\mathcal{D}$, and the functor Q maps formulas to collections of coalgebras.

Coalgebras are described by the endofunctor $B : \mathcal{C} \to \mathcal{C}$. The functor L represents the syntax of the modal operators on the logic. We assume that L has an initial algebra $\alpha : L\Phi \to \Phi$. For example, let $\mathcal{D} = \text{Set}$ and $L = 1 + A \times -$. Our initial algebra then becomes the set A^* of all words over the set A, with algebra structure $\alpha = [\epsilon, \text{cons}] : 1 + A \times A^* \to A^*$.



(2.5)

We also need to define the *semantics* of the logics, which is done by defining a natural transformation $\delta : LP \Rightarrow PB$. We call this map the *one-step semantics* of the logic. This map can be composed with $P\gamma : PBX \to PX$, where (X, γ) is a coalgebra. By initiality of α this gives us a map $[\![-]\!] : \Phi \to PX$ as in diagram 2.6, which denotes the semantics of a logic Φ for a coalgebra (X, γ). This maps a logical formula to a set of states in X that satisfy the formula.

$$\begin{array}{ccc} L\Phi & \xrightarrow{L[\![-]\!]} & LPX \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \delta_X} \\ & & PBX \\ & & \downarrow{\scriptstyle P\gamma} \\ \Phi & \xrightarrow{[\![-]\!]} & PX \end{array} \tag{2.6}$$

Using the duality and the bijection that arises from this duality we can directly construct a map that, given a state in X, gives us the collection of formulas that state satisfies, as a universal property. We call this map $th^\gamma : X \to Q\Phi$ the *theory map* of the coalgebra $\gamma : X \to BX$ for a logic $\alpha : L\Phi \to \Phi$. It is dual to the semantics map through the bijection: $\psi([\![-]\!]) = th$. We can define the theory map inductively using the *mate* [11] of the one-step semantics: $\delta^\flat = QL\epsilon \circ Q\delta Q \circ \eta BQ$, where ϵ and η are the unit and co-unit of the adjunction. Diagram 2.7 shows how the theory map is then inductively expressed as a universal property, the unique morphism making the diagram commute.

$$\begin{array}{ccc} X & \xrightarrow{\exists! th^\gamma} & Q\Phi \\ \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle Q\alpha} \\ & & QL\Phi \\ & & \uparrow{\scriptstyle \delta^\flat_\Phi} \\ BX & \xrightarrow{Bth^\gamma} & BQ\Phi \end{array} \tag{2.7}$$

The following lemma, which will be useful in future proofs, is a corollary of Theorem 3.3 from [11]:

Lemma 2.19. *For coalgebras (X, γ_X) and (M, γ_M) and a coalgebra morphism $f : X \to M$, we have that $th^{\gamma_X} = th^{\gamma_M} \circ f$.*

Proof. In diagram 2.8, the left square commutes because f is a coalgebra morphism, and the right square commutes by diagram 2.7.

$$\begin{array}{ccccc} X & \xrightarrow{f} & M & \xrightarrow{th^{\gamma_M}} & Q\Phi \\ \downarrow{\scriptstyle \gamma_X} & & \downarrow{\scriptstyle \gamma_M} & & \downarrow{\scriptstyle Q\alpha} \\ & & & & QL\Phi \\ & & & & \uparrow{\scriptstyle \delta^\flat_\Phi} \\ BX & \xrightarrow{Bf} & BM & \xrightarrow{Bth^{\gamma_M}} & BQ\Phi \end{array} \tag{2.8}$$

By uniqueness of th^{γ_X} we have that $th^{\gamma_X} = th^{\gamma_M} \circ f$. □

The theory map th^γ_Φ for a coalgebra (X, γ) allows us to identify logically equivalent states, in other words, states which satisfy the same logical

formulas. Expressivity then means that this logical equivalence implies behavioural equivalence. When two processes are behaviourally equivalent, there exists a coalgebra homomorphism between them. Thus, expressivity is defined as follows in [11]:

Definition 2.20. A logic (L, δ) with initial algebra (Φ, α) is *expressive* if for every coalgebra (X, γ) the theory map th^{γ}_{Φ} can be factored into $m \circ e$, where e is a coalgebra homomorphism and m is a monomorphism in $\mathcal{C}$.

The following theorem from [11] gives sufficient conditions for expressivity, which we will use to show expressivity of the logics used in the examples of section 3.3.

Theorem 2.21. *For any endofunctor $B : \mathcal{C} \to \mathcal{C}$ and logic (L, δ) for B-coalgebra's, the logic is expressive if the following conditions are met:*

- *L has an initial algebra*
- *The category $\mathcal{C}$ has a strong epi-mono factorisation system*
- *B preserves monos*
- *$\delta^{\flat} : BQ \Rightarrow QL$ is pointwise monic.*

A coalgebra is minimal with respect to logical equivalence if all its states are uniquely identified by the theory map:

Definition 2.22. A coalgebra (X, γ) is *minimal* w.r.t. Φ if the theory map $th^{\gamma_X}_{\Phi}$ is mono.

There are many types of coalgebraic systems that can be described by an expressive modal logic through this adjunction [10]. In section 3.3 we instantiate the logical framework for two types of coalgebra, Mealy machines and labelled transition systems.

For a simple example, consider the functor $B = 2 \times (-)^A$ in the category of Set. Coalgebras of this functor are deterministic automata with input alphabet A. Let $\mathcal{D} = \mathsf{Set}$ as well and let $P = Q = 2^{(-)}$ be our contravariant functors for the adjunction and let $L = 1 + A \times (-)$. The initial algebra for this functor is the collection of finite and infinite lists of symbols from A, denoted as $A*$, with algebra structure $\alpha = [\epsilon, \mathsf{cons}]$. We define the one-step semantics $\delta_X : LPX \to BPX$ for the two cases of the coproduct: $\delta_X(*) = \{(i, f) \in 2 \times X^A \mid i = 1\}$ and $\delta_X(a, S) = \{(i, f) \in 2 \times X^A \mid f(a) \in S\}$. The semantic map $[\![-]\!]$ can then be retrieved from diagram 2.6: $[\![\epsilon]\!] = \{x \in X \mid \pi_1(\gamma(x)) = 1\}$ and $[\![\mathsf{cons}(a, w)]\!] = \{x \in X \mid \pi_2(\gamma(x))(a) \in [\![w]\!]\}$. The semantic map sends a word w to the collection of states of X that accept w. It is easy to check that the theory map then dually sends a state to the language that is accepted by it.

2.3.2 Subformula Closed Collections of Formulas

Since the theory map from the previous section is only defined on *all* formulas Φ, and since we want our test suite to be finite, we want to be able to somehow restrict the theory map to a smaller collection Ψ. In [4] this is done by defining a property of subobjects of Φ called *subformula closedness*.

Subformula closedness is defined using the notion of *recursive coalgebras* [1]. Let L be an endofunctor in a category $\mathcal{C}$, and let $f : X \to LX$ be a coalgebra. We call σ a recursive coalgebra if for any algebra $g : LY \to Y$ there exists a unique coalgebra-to-algebra map, i.e. a unique morphism such that diagram 2.9 commutes.

$$\begin{array}{ccc} X & \xrightarrow{h} & Y \\ \downarrow{\scriptstyle f} & & \uparrow{\scriptstyle g} \\ LX & \xrightarrow{Lh} & LY \end{array} \tag{2.9}$$

Definition 2.23. A subobject $j : \Psi \rightarrowtail \Phi$ is *subformula closed* if there exists a recursive coalgebra structure $\sigma : X \to LX$ such that j is the unique coalgebra-to-algebra map from $\sigma : \Psi \to L\Psi$ to $\alpha : L\Phi \to \Phi$.

Lemma 10 from [4] then gives us the desired property:

Lemma 2.24. *Let $j : \Psi \to \Phi$ be a subformula closed collection of formulas from Φ, with coalgebra structure $\sigma : \Psi \to L\Psi$. Then $th^{\gamma}_{\Psi} = Qj \circ th^{\gamma}_{\Phi}$ is the unique map such that diagram 2.24 commutes.*

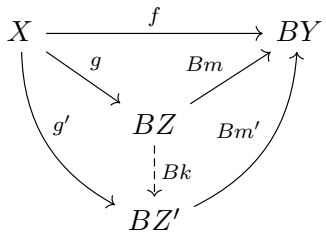
$$\begin{array}{ccccc} X & & \xrightarrow{th^{\gamma}_{\Psi}} & & Q\Psi \\ \downarrow{\scriptstyle \gamma} & & & & \uparrow{\scriptstyle Q\alpha} \\ BX & \xrightarrow{Bth^{\gamma}_{\Psi}} & BQ\Psi & \xrightarrow{\delta^{\flat}_{\Psi}} & QL\Psi \end{array}$$

Lemma 2.24 allows us to define restrictions of the theory map inductively, just like the original theory map.

2.4 Base and Reachability

In order to iteratively run tests on a coalgebra we need to have some method to walk through the states of the coalgebra starting from the initial state x_0. For this we use the notion of the *base* of a morphism, which is taken from [6]. We use the definitions and some examples from [4].

Definition 2.25. Let $\mathcal{C}$ be a category with X, Y objects of $\mathcal{C}$. Let $B : \mathcal{C} \to \mathcal{C}$ be an endofunctor in $\mathcal{C}$. The *base* of a morphism $f : X \to BY$ is a 3-tuple (Z, g, m) with $g : X \to BZ$, and $m : Z \rightarrowtail Y$ a mono, such that $Bm \circ g = f$, and for any (Z', g', m') such that $Bm' \circ g = f$ we have a unique morphism $k : Z \to Z'$ such that diagram 2.10 commutes.



(2.10)

We say that a functor B has a base if every morphism $f : X \to BY$ has a base. From [4] we know that any intersection-preserving functor $B : \mathcal{C} \to \mathcal{C}$ in a well-powered, complete category $\mathcal{C}$ has a base. Since the base of a morphism is unique up-to isomorphism (for two bases (Z, g, m) and (Z', g', m') we get morphisms $k : Z \to Z'$ and $k' : Z' \to Z$) we can call it *the* base of f. The intuition behind the base is that it allows us to talk about the image of a morphism $f : X \to BY$ on the underlying object Y. The base of a morphism under the identity functor Id is just that; the image of the morphism.

For coalgebras the base allows us to talk about direct successors of states. Take a DFA over alphabet A, $\gamma : X \to BX$, $BX = 2 \times X^A$. The base of γ is $Z = \{\pi_2(\gamma(x)(a)) \mid x \in X, a \in A\}$. Then $m : Z \rightarrowtail X$ is the inclusion of Z in X and g is the corestriction of γ on Z: it does exactly the same as γ but its domain BZ contains only the states that are reachable from states in X in one step.

In the context of set endofunctors, we now want to restrict the base to give us the set of successor states reachable in one step from a specific state or set of states. In general this means that, given a subobject $s : S \rightarrowtail X$ of the carrier object X, we want to compute the base of the morphism $\gamma \circ s$. Since the base is another mono, this gives us a new subobject of X. We define an operator $\Gamma : \mathrm{Sub}(X) \to \mathrm{Sub}(X)$ which does exactly this:

Definition 2.26. Let B be a functor with a base, let $\gamma : X \to BX$ be a B coalgebra. Given a subobject $s : S \rightarrowtail X$, let $(\Gamma(S), g, \Gamma(s))$ be the base of the morphism $\gamma \circ s$. See diagram 2.11.

$$\begin{array}{ccc} S & \overset{s}{\rightarrowtail} & X \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle \gamma} \\ B\Gamma(S) & \xrightarrow{B\Gamma(s)} & BX \end{array} \tag{2.11}$$

Intuitively, the base computation can be seen as follows: we start with a machine, which is represented by the coalgebra $\gamma : X \to BX$, but we have access to only some states of X, represented by the subobject $s : S \rightarrowtail X$. When we compute the base, for every state of the machine in s, we get access to every successor of that state. The morphism $g : B\Gamma(S)$ then shows the structure of the transition from the original states to the successor

configurations. The subobject $\Gamma(s) : \Gamma(S) \rightarrowtail X$ then can be seen as a collection of configurations of the machine in all of the successor states of the machine configurations in s. In this way, we can run tests on all of these new states, in different configurations.

When testing a pointed coalgebra (X, γ, x_0) using our algorithm we will repeatedly apply this Γ-operator, giving us a sequence of subobjects $s_i : S_i \rightarrowtail X$ with $s_0 = x_0$ and $s_{i+1} = \Gamma(s_i) : \Gamma(S_i) \rightarrowtail X$. In a reachable and finite system the union of this sequence of subobjects should be isomorphic to X. Reachability for a pointed coalgebra is defined as follows:

Definition 2.27. A coalgebra (X, γ, x_0) is *reachable* if for any subobject $s : S \rightarrowtail X$ with $s_0 : 1 \rightarrowtail S$ such that $s_0 = s \circ x_0$ we have that if S is a subcoalgebra of (X, γ) then s is an isomorphism.

In [4] the following theorem provides an alternative notion of reachability, where operator $\Gamma \vee x_0$ is applied iteratively until the entire coalgebra is reached.

Theorem 2.28. *Let B be an endofunctor with a base, let (X, γ, x_0) be a pointed B-coalgebra. Then X is reachable if and only if $X \cong lfp(\Gamma \vee x_0)$.*

Chapter 3

Coalgebra Testing

We have seen that for deterministic Mealy machines tests are intuitively defined as sequences of input, and the result of a test is a sequence of outputs. For different types of state-based systems, both the type of test and the kind of result you get can differ wildly. For example, while Mealy machines describe a sequence transformation, deterministic finite automata (DFA) describe regular languages. So while the tests are also sequences of inputs, the test result is a yes/no answer.

For nondeterministic systems, such as labelled transition systems (LTS), tests become more complicated. To account for their branching behaviour, we use formulas with modal operators combined with input symbols. A test result is then an evaluation of the formula in the state that is being tested. Note that, for a black-box system, it is not obvious that this is possible as in reality you can't guarantee that even in a finite system all branches will be evaluated. Previous work on testing nondeterministic Mealy machines assumes a certain fairness in implementation, that after sufficient testing we can have confidence that we did not miss any states [17, 20].

Our approach assumes access to the *base* [6] of a functor, which essentially computes all immediate next states from a given states. This is comparable to the fairness assumption: intuitively, computation of the base can be interpreted as the ability to set the conditions of an implementation in such a way that we can eventually access any state.

Whereas usually test-suites consists of both a part that traverses all states and transitions, and a part that distinguishes states in the implementation with respect to inequivalent states in the model, our test suite (the actual collection of tests) consists of only the second part, where the first task is managed by the base.

We will first define properly what a test suite is in section 3.1, and the specific properties that our test suite will have to have in order for it to be n-complete. We will then define exactly what n-completeness means for coalgebras. In section 3.2 we will give the testing algorithm, and prove that,

if you run the algorithm with a test suite that has the desired properties, it will be n-complete. We end this chapter with two elaborated example cases, for Mealy machines and labelled transition systems, in section 3.3.

3.1 Definitions

The definition of a test suite is simple:

Definition 3.1. A *test suite* is a subformula-closed collection of formulas $\Psi \rightarrowtail \Phi$ of some logic Φ.

In our test method, the test suite serves only to *distinguish* states in the implementation from states that are represented by inequivalent states in the model. The test suite is subformula closed so we can restrict the theory map to the test suite when evaluating.

Definition 3.2. A subformula-closed collection of formulas is a *characterisation* of a coalgebra (X, γ) if for every pair of parallel morphisms $s : S \to X$ and $s' : S \to X$ we have that $th^{\gamma}_{\Psi} \circ s = th^{\gamma}_{\Psi} \circ s'$ implies that $th^{\gamma}_{\Phi} \circ s = th^{\gamma}_{\Phi} \circ s'$.

The characterisation allows us to separate logically inequivalent states using a smaller (and, for n-completeness, finite) collection of tests. This definition states that if two subobjects of the state space of a coalgebra can be separated using the logic Φ, then they can be separated using the collection Ψ, a subobject of the entire logic. This is analogous to the characterisation set for testing Mealy machines: where the entire logic is the set of all input sequences, the characterisation is a set of inputs that distinguishes all inequivalent states. Since this logic is expressive, behavioural inequivalence coincides with logical inequivalence.

Because our method of accessing states and distinguishing states are so separated, the notion *passing* a set of tests becomes slightly less obvious. The first part of our method "connects" states in the implementation to their supposed representative states in the model. When we assume a minimal model, this connection should be a homomorphism up-to logical equivalence from the implementation coalgebra to the specification coalgebra. The characterisation then checks whether this connection is actually such a homomorphism up-to logical equivalence.

Definition 3.3. Given coalgebras (X, γ_X) and (M, γ_M), a morphism $f : X \to M$ is a coalgebra homomorphism up-to logical equivalence w.r.t. a subformula-closed collection of formulas $\Psi \rightarrowtail \Phi$ if the diagram 3.1 commutes.

$$\begin{array}{ccccc} X & \xrightarrow{f} & M & \xrightarrow{\gamma_M} & BM \\ \downarrow{\scriptstyle \gamma_X} & & & & \downarrow{\scriptstyle Bth^{\gamma_M}_{\Psi}} \\ BX & \xrightarrow{Bf} & BM & \xrightarrow{Bth^{\gamma_M}_{\Psi}} & BQ\Psi \end{array} \tag{3.1}$$

If Ψ is a test suite for $(M, \gamma_M.m_0)$ then an intuitive notion of an implementation (X, γ_X, x_0) passing Ψ would be if the initial states have equal test results. However, since in our test method we don't include the reachability part inside the test suite, we need to specify that an implementation passes the test suite not only when its initial state is equivalent to the model's initial state with respect to Ψ, but any successor state in X should be equivalent to the corresponding state in M with respect to Ψ.

Definition 3.4. A implementation (X, γ_X, x_0) *passes* a test suite $\Psi \rightarrowtail \Phi$ for a specification $(M, \gamma_M.m_0)$ if there exists a homomorphism up-to logical equivalence w.r.t. Ψ $h : X \rightarrowtail M$ such that $h \circ x_0 = m_0$.

The homomorphism in definition 3.4 is built in our testing algorithm by iteratively testing states and then accessing their successors through the computation of the base.

Since with testing we talk about n-completeness, where implementations are limited in their reachable number of states, we want to translate this property to finite systems. In that case, the fixed point in theorem 2.28 should be reached after a finite number of applications. This gives us a nice notion of systems that are reachable within n in steps outside of categories where we can talk about the number of elements of an object.

Definition 3.5. Let B be an endofunctor with a base, and let (X, γ, x_0) be a pointed B-coalgebra. A pointed B-coalgebra. Let $\{s_i : S_i \rightarrowtail X\}$ be a sequence of subobjects such that $s_0 = x_0$ and $s_{i+1} = \Gamma(s_i)$. Then we call (X, γ, x_0) *n-reachable* if $\bigvee_{0 \leq i \leq n} \{s_i : S_i \rightarrowtail X\} \cong X$.

Since all of the examples in this thesis will be coalgebras in the category of sets and functions, we show that a reachable coalgebra with at most n number of states is n-reachable:

Proposition 3.6. *Let $B : Set \to Set$ be an endofunctor on Set. Let (X, γ_X, x_0) be a reachable pointed B-coalgebra with $|X| \leq n$ states. Then (X, γ_X, x_0) is n-reachable.*

Proof. Let $S_0 = 1$, $s_0 : S_0 \to X$ be defined as $s_0 = x_0$. Let $(S_{i+1}, g_i, s_{i+1}) = (\Gamma(S_i), g_i, \Gamma(s_i))$ be the base of $\gamma \circ s_i$ as in definition 3.5. Let $k_i : S_i \rightarrowtail \bigvee_{i \leq n} S_i$ be the inclusion of S_i into the join of $\{s_i : S_i \rightarrowtail X\}_{i \leq n}$. We know that the meet $\{o_i : \bigwedge_{i \leq n} \rightarrowtail S_i\}_{i \leq n}$ is the pullback of the collection of subobjects $\{s_i : S_i \rightarrowtail X\}_{i \leq n}$, with o_i the inclusion of $\bigwedge_{i \leq n}$ into S_i. We know the join is the pushout of diagram $\{o_i : \bigwedge_{i \leq n} \rightarrowtail S_i\}_{i \leq n}$, so by the universal property of the pushout, we get a morphism $s : \bigvee_{i \leq n} S_i \rightarrowtail X$ such that $s \circ k_i = s_i$. Since $|X| = n$, by monotonicity of Γ we have $\bigvee_{i \leq n+1} S_i = \bigvee_{i \leq n} S_i$. We need this because we want to build a subcoalgebra $g : \bigvee_{i \leq n} S_i \to B \bigvee_{i \leq n} S_i$.

Now consider a collection of morphisms $\{Bk_{i+1} \circ g_i : S_i \to B \bigvee_{i \leq n} S_i\}$. In order to get g via the universal property of the pushout $k_i : S_i \rightarrowtail \bigvee_{i \leq n} S_i$ we

want to show that for any $i, j \leq n$ we have that $Bk_{i+1} \circ g_i \circ o_i = Bk_{j+1} \circ g_j \circ o_j$. Since s is monotone (pushouts conserve monotonocity in Set), we do this by proving $Bs \circ Bk_{i+1} \circ g_i \circ o_i = Bs \circ Bk_{j+1} \circ g_j \circ o_j$.

We have that $s \circ k_{i+1} = s_{i+1}$ for all i, so $Bs_{i+1} \circ g_i \circ o_i = Bs_{j+1} \circ g_j \circ o_j$. Then by definition of g_i we have that $\gamma \circ s_i \circ o_i = \gamma \circ s_j$. Because we know $\{o_i : \bigwedge_{i \leq n} \rightarrowtail S_i\}_{i \leq n}$ is the pullback of $\{s_i : S_i \rightarrowtail X\}_{i \leq n}$, we have that $s_i \circ o_i = s_j \circ o_j$. Now again by the universal property of the pushout we get a unique map $g : \bigvee_{i \leq n} S_i \to B \bigvee_{i \leq n} S_i$.

The final collection of morphisms to consider is $\{Bs_{i+1} \circ g_i : S_i \to BX\}_{i \leq n}$. For $i, j \leq n$ we have that $Bs_{i+1} \circ g_i \circ o_i = Bs \circ Bk_{i+1} \circ g_i \circ o_i = Bs \circ g \circ k_i \circ o_i = Bs \circ g \circ k_j \circ o_j$. So yet again by the universal property $Bs \circ g$ is the unique morphism such that $Bs \circ g \circ k_i = Bs_{i+1} \circ g_i$. But we also have that $Bs_{i+1} \circ g_i = \gamma \circ s_i = \gamma \circ s \circ k_i$. Then by uniqueness of $Bs \circ g$ we have that $\gamma \circ s = Bs \circ g$. This means g is a subcoalgebra of γ, and by reachability of γ this means that s is an isomorphism. Thus, $\bigvee_{i \leq n} \cong X$. □

Using these definitions, we can define n-completeness in a relatively intuitive way:

Definition 3.7. A test suite $\Psi \rightarrowtail \Phi$ is *n-complete* for a specification (S, γ_S) if any logically inequivalent, m-reachable implementation (X, γ_X) with $m \leq n$ does not pass the test suite.

3.2 Test Method

3.2.1 Setting

In this section we will show that a characterisation Ψ of a specification (M, γ_M, m_0) is an n-complete test suite for M. We do this by proposing an algorithm that, given an implementation machine (X, γ_X, x_0), tries to build a morphism as in definition 3.4. The algorithm should succeed if (X, γ_X, x_0) is logically equivalent to (M, γ_M, m_0), and fail otherwise. The computation of the base allows us to traverse the implementation step-by-step. Since we assume that the implementation has at most n states, we need at most n base computations before we have visited every state in the implementation. States accessed by the base are then tested using the characterisation Ψ of the model.

We assume our specification to minimal with respect to logical equivalence, as having the theory map be a monomorphism significantly simplifies some proofs and helps limit the scope of this thesis somewhat. However, minimality with respect to *any* logic implies minimality with respect to behavioural equivalence. This makes sense, as behavioural equivalence is as strong as any logical equivalence, and if two states are already separated by some weaker notion of equivalence, then they are definitely separated by behavioural equivalence.

This has as a consequence that any homomorphism up-to logical equivalence (definition 3.3) becomes a regular coalgebra homomorphism if Ψ is a characterisation. As such, the requirement that the model is minimal with respect to logical equivalence has as a consequence that our algorithm as it is can only test for behavioural equivalence. Because of this, we will assume that our logic is expressive for the rest of this chapter. Another argument for only using expressive logic is the base; since we test each successor separately, for every successor of the initial state in X there should be a successor of the initial state in M that is logically equivalent and reached by the same transition. This is not the case for every logic, e.g. trace equivalence for labelled transition systems [11].

Concretely, we assume the following:

- A complete, well-powered and adhesive category $\mathcal{C}$
- A functor $B : \mathcal{C} \to \mathcal{C}$ that preserves wide meets and joins
- A functor $L : \mathcal{D} \to \mathcal{D}$ with initial algebra $\alpha : L\Phi \to \Phi$
- An adjunction $P \dashv Q : \mathcal{C} \leftrightarrows \mathcal{D}^{\mathsf{op}}$, with one-step semantics $\delta : LP \Rightarrow PB$
- A minimal coalgebra (M, γ_M, m_0), which we will refer to as the specification or model
- A implementation coalgebra (X, γ_X, x_0)
- A characterisation $j : \Psi \rightarrowtail \Phi$ of the specification (M, γ_M, m_0).

We access the implementation machine only through the computation of the base of the composition $\gamma_X \circ s$ of γ_X with any subobject $s : S \rightarrowtail X$, and using the theory map $th_\Psi^{\gamma_X}$ for testing.

3.2.2 Algorithm

In order to connect the states we get from the base to states in the model we need some procedure. The way these should be connected depends on the structure of the functor B. For example, let $B = 2 \times X^A$ be the functor for DFA's. We have a subobject $s : S \rightarrowtail X$ and a morphism $h : S \to M$. Taking the base of the morphism $\gamma_X \circ s$ gives us $g : S \to 2 \times \Gamma(S)^A$. The structure of g with respect to S is the same as the structure of the system under test γ_X. For example, let $x \in S$ and let $\pi_2(g(x))(a) = y \in \Gamma(S)$. We want to send y to the state $m \in M$ such that $\pi_2(\gamma_M(h(x)))(a) = m$.

Testing can already fail here: for DFAs, a nonfinal state can have been linked to a final state. In the case of Mealy machines there might not be a transition with a matching input/output combination in the specification. For labelled transition systems, there might be multiple candidate states in

M for a state y in $\Gamma(S)$. To abstract away from this for all these different systems, we define the set $\text{link}(S)$, which contains all possible candidate morphisms:

Definition 3.8. We define the set $\mathit{link}(g, h)$ to be the set of morphisms h' that satisfy diagram 3.2.

$$\begin{array}{ccc} S & \xrightarrow{h} & M \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle \gamma_M} \\ B\Gamma(S) & \xrightarrow{Bh'} & BM \end{array} \tag{3.2}$$

We need this set every iteration of the algorithm; if such a set cannot be constructed, then the algorithm fails. As such, we need a lemma to ensure that this does not happen when we're testing a machine equivalent to our specification:

Lemma 3.9. *If (M, γ_M, m_0) and (X, γ_X, x_0) are equivalent and $h \leq f$ (so $h = f \circ s$) with f a homomorphism, then $\mathit{link}(g, h)$ is nonempty. Specifically, $h' = f \circ \Gamma(s) \in \mathit{link}(g, h)$*

Proof. Take $h' = f \circ \Gamma(s)$. Then $h' \circ g = Bf \circ B\Gamma(s) \circ g = Bf \circ \gamma_X \circ s$ by the base, and $Bf \circ \gamma_X \circ s = \gamma_M \circ f \circ s = \gamma_M \circ h$ because f is a homomorphism. Expanding diagram 3.2 in diagram 3.3 shows that the diagram commutes.

$$\begin{array}{ccccc} S & \xrightarrow{s} & X & \xrightarrow{f} & M \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle \gamma_X} & & \downarrow{\scriptstyle \gamma_M} \\ B\Gamma(S) & \xrightarrow{B\Gamma(s)} & BX & \xrightarrow{Bf} & BM \end{array} \tag{3.3}$$

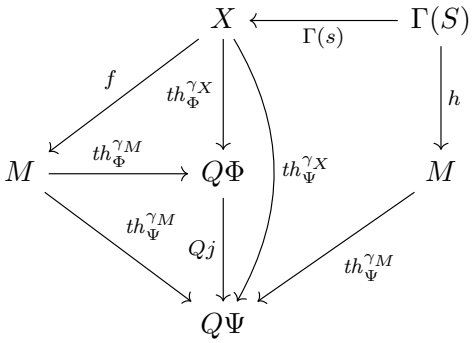
□

In the algorithm we build a coalgebra homomorphism from X to M by defining a part of it as a morphism on a subobject of X each iteration. In the following lemma we show that if we can construct such a morphism, and it is admitted by Ψ, then it is under a homomorphism f.

Lemma 3.10. *Let (X, γ_X) and (M, γ_M) be coalgebra, let M be minimal w.r.t. to logic Φ. Let $f : X \to M$ be a coalgebra morphism. Let $j : \Psi \rightarrowtail \Phi$ be a characterisation of M. Then if $th_\Psi^{\gamma_X} \circ \Gamma(s) = th_\Psi^{\gamma_M} \circ h$ for $h : \Gamma(S) \to M$, we have that $h = f \circ \Gamma(s)$.*

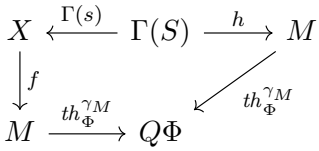
Proof. In diagram 3.4, the square on the right commutes by assumption, the upper left triangle commutes by lemma 2.19, and the other two inner

triangles commute because Ψ is a subformula closed collection of formula.



(3.4)

Because Ψ is a characterisation, the outer commutation gives us:



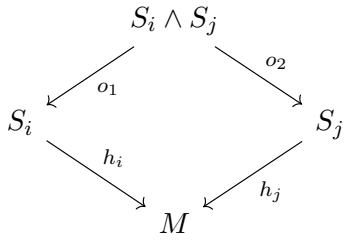
(3.5)

Because M is minimal w.r.t. Φ the theory map $th_\Phi^{\gamma_M}$ is mono, and thus we get that $h = f \circ \Gamma(s)$. □

If we assume that the coalgebra under test (X, γ_X, x_0) and the model (M, γ_M, m_0) are behaviourally equivalent then there exists some coalgebra homomorphism $f : X \to M$ such that $f(x_0) = m_0$. Thus we can build a sequence of morphisms starting with $h_0 : S_0 \to M$ with $S_0 = 1$ and $s_0 : 1 \to X$ equal to x_0. and $h_0 = m_0$. We have that $h_0 = f \circ s_0$. Let $S_{i+1} = \Gamma(S_i)$ with $s_{i+1} = \Gamma(s_i)$. By lemmas 3.9 and 3.10 we get a sequence $\{h_i : S_i \to M\}_i$ such that $h_i = f \circ s_i$. We can use this sequence to 'build' the homomorphism f using the fact that X is reachable, since the union $\{s_i : S_i \to X\}$ should be isomorphic to X, $\cup_i s_i : S_i \to X \cong X$ as stated in proposition 3.6. This ensures that our algorithm will always succeed on a machine that is equivalent.

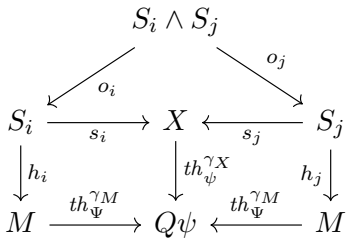
Then we can construct a morphism h by using the fact that the union of subobject is a (wide) pushout. The universal property and proposition 3.6 then gives us a morphism $h : X \to M$. Before that, we need the following lemma:

Lemma 3.11. *Let $\{s_i : S_i \rightarrowtail X\}$ be a sequence of subobjects of X, where $s_0 = x_0 : 1 \rightarrowtail X$ and $s_{i+1} = \Gamma(s_i) : \Gamma(S_i) \rightarrowtail X$. Let $\{h_i : S_i \to M\}$ be morphisms such that $th_\Psi^{\gamma_M} \circ h_i = th_\Psi^{\gamma_X} \circ s_i$. For any pair of subobjects S_i and S_j of X, $S_i \wedge S_j$ the meet of S_i and S_j and $o_i : S_i \wedge S_j \rightarrowtail S_i$, $o_j : S_i \wedge S_j \rightarrowtail S_j$ the inclusion maps, the following diagram 3.6 commutes:*



(3.6)

Proof. Consider the following diagram (3.7):



(3.7)

We have that the upper triangle commutes because $S_i \wedge S_j$ is the pullback of s_i and s_j. The squares on the bottom commute because of the testing we do during the algorithm. Since Ψ is a characterisation of M, $th_{\Psi}^{\gamma_M} \circ h_i \circ o_i = th_{\Psi}^{\gamma_M} \circ h_j \circ o_j$ implies $th_{\Phi}^{\gamma_M} \circ h_i \circ o_i = th_{\Phi}^{\gamma_M} \circ h_j \circ o_j$, and by minimality of M with respect to Φ its theory map is mono, so we get the required result. □

Now that we have defined all the moving parts and proven lemmas to show that each part functions properly, we can define algorithm 1. The algorithm computes the base of $\gamma_X \circ s_i$ at the start of every iteration i, and tests all morphisms in the set $\text{link}(g_i, h_i)$ (definition 3.8). If this set is empty, or there is no morphism that is admitted by Ψ, the algorithm fails. Otherwise, it continues onto the next iteration, until all states in X should have been reached.

Algorithm 1 The testing algorithm

$s_0 : S_0 \to X \leftarrow x_0 : 1 \to X$
$h_0 \leftarrow m_0$
for $i = 0$ **to** $n - 1$ **do**
  let $(\Gamma(S_i), g_i, \Gamma(s_i))$ be the B-base of $\gamma_X \circ s_i : S_i \rightarrowtail X$
  $S_{i+1} \leftarrow \Gamma(S_i)$
  $s_{i+1} \leftarrow \Gamma(s_i)$
  pass $\leftarrow$ false
  for all $h \in \text{link}(g_i, h_i)$ **do**
    if $th_\Psi^{\gamma_X} \circ \Gamma(s_i) = th_\Psi^{\gamma_M} \circ h$ **then**
      $h_{i+1} \leftarrow h$
      pass $\leftarrow$ true
      continue
    end if
  end for
  if not pass **then**
    return fail
  end if
end for
return success

3.2.3 Soundness and Completeness

Soundness in terms of testing means that if a implementation has no errors then the algorithm will not return an error. For pointed coalgebras, an implementation without errors means that there is a coalgebra morphism between the implementation and the model which connects the initial states. We prove this in the following theorem:

Theorem 3.12. *Given a coalgebra (M, γ_M) that is minimal w.r.t. logic Φ serving as the model, with characterisation Ψ, an implementation coalgebra (X, γ_X), and a coalgebra morphism $f : X \to M$ such that $f \circ x_0 = m_0$, then algorithm 1 will terminate successfully.*

Proof. In order for algorithm 1 to terminate successfully, the variable 'pass' has to be set to true every iteration of the outer for-loop. For this to happen in iteration i, we need to find a $h \in \text{link}(g, \gamma_M \circ h_i)$ which passes Ψ: $th_\Psi^{\gamma_X} \circ \Gamma(s_i) = th_\Psi^{\gamma_M} \circ h$. We do this by showing that in every iteration i we find that $h_i = f \circ s_i$. By lemma 2.19 we then get that h passes (since $h = h_{i+1} = f \circ s_{i+1} = f \circ \Gamma(s_i)$. For $i = 0$ we have that $h_0 = m_0$ and $s_0 = x_0$,so $h_0 = f \circ x_0 = f \circ s_0$.

Let $h_i = f \circ s_i$. Then by lemma 3.9 we have that $f \circ \Gamma(s_i) \in \text{link}(g_i, h_i)$. By lemma 2.19 we get that $th_\Psi^{\gamma_X} \circ \Gamma(s_i) = th_\Psi^{\gamma_M} \circ h$ for $h = f \circ \Gamma(s_i)$. So there is at least one $h \in \text{link}(g_i, h_i)$ which passes the tests Ψ. Lemma 3.10

shows that testing filters out any other h in $\text{link}(g_i, h_i)$. Thus we get that $h_{i+1} = f \circ \Gamma(s_i) = f \circ s_{i+1}$.

By induction we have that this holds for all i and in particular for all $i \leq n$, thus the algorithm terminates successfully. □

Completeness for testing means that every error is detected, that is, for any erroneous implementation (X, γ_X) algorithm 1 fails. In other words, if the algorithm is successful then implementation X and model M are equivalent.

Theorem 3.13. *Given a coalgebra (M, γ_M) that is minimal w.r.t. logic Φ, serving as the model, and an n-reachable implementation coalgebra (X, γ_X), if algorithm 1 terminates successfully, then M and X are behaviourally equivalent. That is, there exists a coalgebra morphism $f : X \to M$.*

Proof. If the algorithm terminates, we have a set of morphisms $\{h_i : S_i \to M\}_{i \leq n}$. By using the fact that the union of subobjects $\bigvee_{i \leq n} S_i$ is a wide pushout of $\{o_i : \bigwedge_{i \leq n} S_i \to S_i\}$ with o_i the projections from the product, and the fact that X is isomorphic to this union by assumption of n-reachability, by the UMP of the pushout we get a unique morphism $h : X \to M$ such that, for all i, $h \circ s_i \circ o_i = h_i \circ o_i$.

We want to prove that this h is an coalgebra morphism. Again by the UMP, we get that $\gamma_M \circ h$ is the unique morphism such that for all i, $\gamma_M \circ h \circ s_i \circ o_i = \gamma_M \circ h_i \circ o_i$.

By diagram 3.2 from the definition of $\text{link}(g_i, h_i)$ we have that $\gamma_M \circ h_i = Bh_{i+1} \circ g_i$. Then again we have $Bh_{i+1} \circ g_i = Bh \circ Bs_{i+1} \circ g_i$ by definiton of h, and by definition of the base we have $Bs_{i+1} \circ g_i = \gamma_X \circ s_i$. So for all i we have that $Bh \circ \gamma_X \circ s_i = \gamma_M \circ h_i$, and in particular $Bh \circ \gamma_X \circ s_i \circ o_i = \gamma_M \circ h_i \circ o_i$.

By uniqueness, we thus get that $\gamma_M \circ h = Bh \circ \gamma_X$. Thus, h is a coalgebra homomorphism making X and M behaviourally equivalent. □

3.3 Examples

3.3.1 Mealy Machines

In this section we apply the algorithm to coalgebras representing Mealy machines. That is, coalgebras for the endofunctor $BX = (O \times X)^A$ in Set, where A is a set of input symbols and O is the set of output symbols. For our logic we take the functor $L = A \times (1 + -)$. The initial algebra for this functor is A^+, the set of finite nonempty words over A. We need the words to be nonempty since output is defined on the transitions rather than the states, as in a DFA. First, we should set up the logical framework as in

section 2.3, in order to acquire the theory map.

$$\begin{aligned} P &= Q = O^- \\ B &= (O \times -)^A \\ L &= A \times (1 + -) \\ \Phi &= A^+ \\ \delta &: A \times (1 + O^-) \Rightarrow O^{B-} \\ \delta_X(a, \mathrm{inl}(*)) &= \lambda f.\pi_1(f(a)) \\ \delta_X(a, \mathrm{inr}(g)) &= \lambda f.g(\pi_2(f(a))) \end{aligned}$$

$O^- : \mathrm{Set} \to \mathrm{Set}^{\mathrm{op}}$ is the contravariant functor that sends a set X to the set of functions $\{f : X \to O\}$, and sends a function $f : X \to Y$ to a function $O^f : O^Y \to O^X$ in the following way: $O^f(g) = g \circ f$. Our adjunction is then $O^- \dashv O^- : \mathrm{Set} \leftrightarrows \mathrm{Set}^{\mathrm{op}}$. with η and ϵ the unit and counit of the adjunction, defined by $\eta_X(x) = \epsilon_X(x) = \lambda f.f(x)$.

Filling in the one-step semantics δ_X into diagram 2.6 gives us the following two equations for the semantics map:

$$\begin{aligned} [\![a]\!](x) &= \pi_1(\gamma(x)(a)) \\ [\![aw]\!](x) &= [\![w]\!](\pi_2(\gamma(x)(a))) \end{aligned}$$

The semantics for a word $w \in A^+$ at state x thus is the last output symbol after giving w as input in state x. This explains why we take our tests from A^+ instead of A^* as in the concrete case (section 2.1); in the concrete case, test results are sequences from O^* rather than individual symbols. As such, the empty word as a test results in the empty word as a test result, in any Mealy machine, so any machine passes the empty word. In the coalgebraic case, output is undefined on the empty word, as we need to produce at least one output symbol. This difference is a mere technicality however, as we will show that our testing process uses the same characterisation as in section 2.1.3.

To get the theory map we first have to calculate the mate of the one-step

semantics, $\delta^\flat : BQ \Rightarrow QL$, given by $\delta^\flat = QL\epsilon \circ Q\delta Q \circ \eta BQ$.

$$\begin{aligned}
\delta^\flat_\Phi(f)(a, \mathrm{inl}(*)) &= (QL\eta \circ Q\delta Q \circ \eta BQ)(a, \mathrm{inl}(*)) \\
&= (QL\eta \circ Q\delta Q(\lambda g.g(f)))(a, \mathrm{inl}(*)) \\
&= (QL\eta \circ ((\lambda g.g(f)) \circ \delta_{Q\Phi}))(a, \mathrm{inl}(*)) \\
&= (\lambda g.g(f))(\delta_{Q\Phi}(a, \mathrm{inl}(*))) \\
&= \delta_{Q\Phi}(a, \mathrm{inl}(*))(f) \\
&= (\lambda f'.\pi_1(f'(a)))(f) \\
&= \pi_1(f(a)) \\
\delta^\flat_\Phi(f)(a, \mathrm{inr}(w)) &= (QL\eta(\lambda g.g(f)) \circ \delta_{Q\Phi})(a, \mathrm{inr}(w)) \\
&= (\lambda g.g(f))(\delta_{Q\Phi}(a, \mathrm{inr}(\eta(w)))) \\
&= (\lambda g.g(f))(\delta_{Q\Phi}(a, \mathrm{inr}(\lambda f'.f'(w)))) \\
&= (\lambda g.g(f))(\lambda f''.(\lambda f'.f'(w))(\pi_2(f''(a)))) \\
&= (\lambda f''.(\lambda f'.f'(w))(\pi_2(f''(a))))(f) \\
&= (\lambda f'.f'(w))(\pi_2(f(a))) \\
&= \pi_2(f(a))(w)
\end{aligned}$$

Filling this in into diagram 2.7 gives equation:

$$Q\alpha \circ th^\gamma = \delta^\flat_\Phi \circ Bth^\gamma \circ \gamma$$

Concretely, we get two equations, one for each side of the coproduct:

$$\begin{aligned}
Q\alpha(th^\gamma(x))(a, \mathrm{inl}(*)) &= (\delta^\flat_\Phi \circ Bth^\gamma \circ \gamma)(x)(a, \mathrm{inl}(*)) \\
&= \pi_1(Bth^\gamma(\gamma(x))(a)) \\
&= \pi_1(\pi_1(\gamma(x)(a)), th^\gamma(\pi_2(\gamma(x)(a)))) \\
&= \pi_1(\gamma(x)(a)) \\
Q\alpha(th^\gamma(x))(a, \mathrm{inr}(w)) &= (\delta^\flat_\Phi \circ Bth^\gamma \circ \gamma)(x)(a, \mathrm{inr}(w)) \\
&= \pi_2(\pi_1(\gamma(x)(a)), th^\gamma(\pi_2(\gamma(x)(a))))(w) \\
&= th^\gamma(\pi_2(\gamma(x)(a)))(w)
\end{aligned}$$

Simplifying the left side this becomes:

$$\begin{aligned}
th^\gamma(x)(a) &= \pi_1(\gamma(x)(a)) \\
th^\gamma(x)(aw) &= th^\gamma(\pi_2(\gamma(x)(a)))(w)
\end{aligned}$$

This defines the theory map inductively in such a way that every state is mapped to a function which then maps an input w to the last output symbol you get if you start inputting w at state x. Now that we have the definition of the theory map, we still need this logic to be expressive. Recall that Theorem 2.21 gives sufficient conditions for expressivity.

The first three conditions clearly hold in this example. What's left for us to prove is pointwise monicity of the mate of the one-step semantics. This is also quite trivial; let $f, g \in BQ\Phi$ such that for all $a \in A$:

- $\delta^\flat_\Phi(f)(a, \mathrm{inl}(*) = \delta^\flat_\Phi(g)(a, \mathrm{inl}(*))$
- for all $w \in \Phi$, $\delta^\flat_\Phi(f)(a, \mathrm{inr}(w) = \delta^\flat_\Phi(g)(a, \mathrm{inr}(w))$

Then for all $a \in A$ we have that $\pi_1(f(a)) = \pi_1(g(a))$, and for all $a \in A$, $w \in \Phi$ we have that $\pi_2(f(a))(w) = \pi_2(g(a))(w)$. Thus for all $a \in A$, $f(a) = g(a)$ and $\delta^\flat_\Phi$ is monic.

Now that we've set up our logical framework, we can start setting up the testing process. Our specification Mealy machine will be the one in figure 3.1, the machine as in the example for the classical testing method in section 2.1.3.

As in the example of 2.1.3, we need a characterisation for this Mealy machine, as in definition 3.2. We can actually take same the set $\Psi = W = \{aa, a, c, ac\}$, since, as show in the classical example, all of the last symbols in the output are distinguishing.

Now we know that if two states in the specification, which we will call (M, γ_M, m_0) or just M from now on, give equal outputs for all inputs in Ψ, they are equal (and thus logically equivalent). This shows that M is minimal with respect to the logic (since every state can be separated from the other states) and by expressivity it is also minimal with respect to behavioural equivalence. Thus we will take Ψ as our characterisation of the minimal specification machine M.

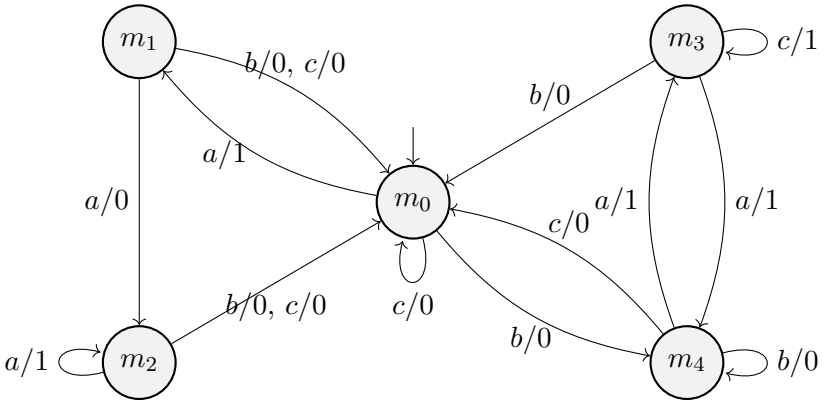


Figure 3.1: Specification Mealy machine

We will run the testing algorithm using the characterisation Ψ on two implementation machines (X, γ_X, x_0) and (Y, γ_Y, y_0), illustrated in figure 3.2 and 3.3 respectively. The first implementation, X, is correct, but the second implementation, Y, contains an error.

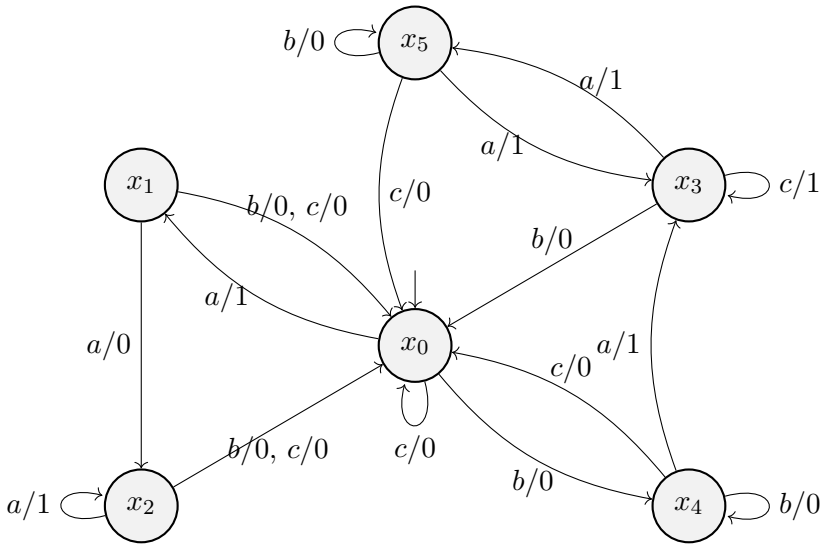


Figure 3.2: Correct implementation (X, γ_X, x_0) of specification (M, γ_M, m_0) (figure 3.1)

First, we show the testing process for the correct implementation (X, γ_X, x_0). This implementation has an extra state s_5, which is equivalent to s_4 and is reachable through s_3. We will walk through Algorithm 1 step-by-step, beginning by initialising the first subobject $s_0 : S_0 \to X$ as $x_0 : 1 \to X$ and the first morphism $h_0 : S_0 \to M$ as $m_0 : 1 \to M$ connecting the subobject of X to M.

We then compute the subsequent subobjects S_i and morphisms $h_i : S_i \to M$ in the for-loop by iterative base computation and testing. First, we compute the base of $\gamma_X \circ s_0$. For Mealy machines the base $(\Gamma(S), g, \Gamma(s))$ of a morphism $\gamma \circ s$ is given by $\Gamma(s) : \Gamma(S) \rightarrowtail X$ where $\Gamma(S) = \{\pi_2(\gamma_X \circ s(x)(a)) | x \in S, a \in A\}$, or something isomorphic to this set. $\Gamma(s)$ is the inclusion in X and g is the corestriction of $\gamma \circ s$ to $B\Gamma(S)$. Note that we cannot actually see the 'names' of elements of X. The base just models a way to access states for testing. As such, they will never actually be used for identification, but we will call the states by their name for clarity in this example.

For the first iteration, the base $(\Gamma(S_0), g_0, \Gamma(s_0))$ is as follows:

$$\begin{aligned} \Gamma(S_0) &= \{x_0, x_1, x_4\} \\ g_0(*)(a) &= (1, x_1) \\ g_0(*)(b) &= (0, x_4) \\ g_0(*)(c) &= (0, x_0) \end{aligned}$$

Now we have to compute h_1. In our abstract algorithm we get a set of morphisms $\text{link}(g_0, h_0)$ such that for all $h \in \text{link}(g_0, h_0)$ $Bh \circ g_0 = \gamma_M \circ h_0$.

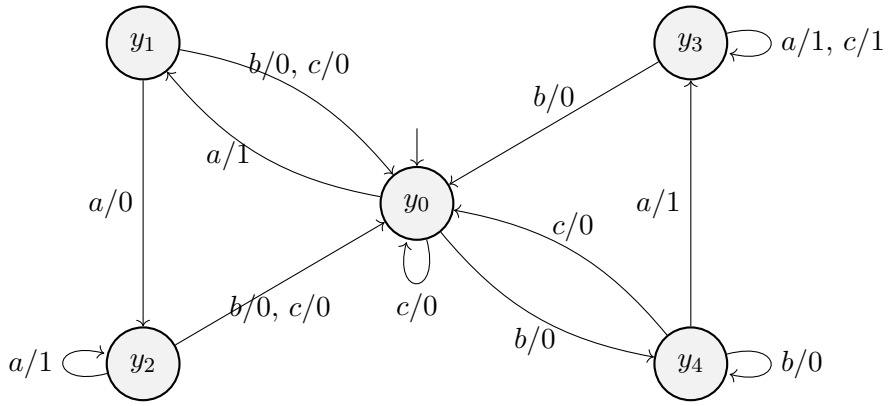


Figure 3.3: Faulty implementation (Y, γ_Y, y_0) of specification (M, γ_M, m_0) (figure 3.1)

.

For the deterministic Mealy machines the morphism that satisfies this is unique:

$$(1, h_1(x_1)) = Bh_1 \circ g_0(*)(a) = \gamma_M \circ h_0(*)(a) = \gamma_M(m_0)(a) = (1, m_1)$$
$$(0, h_1(x_4)) = Bh_1 \circ g_0(*)(b) = \gamma_M \circ h_0(*)(b) = \gamma_M(m_0)(b) = (0, m_4)$$
$$(0, h_1(x_0)) = Bh_1 \circ g_0(*)(c) = \gamma_M \circ h_0(*)(a) = \gamma_M(m_0)(a) = (0, m_0)$$

Morphism $h_1 : S_1 \to M$ should then be defined as follows:

$$h_1(x_1) = m_1$$
$$h_1(x_4) = m_4$$
$$h_1(x_0) = m_0$$

Finally, we 'run the tests', checking whether $th_\Psi^{\gamma_X} \circ \Gamma(s_0) = th_\Psi^{\gamma_M} \circ h_1$ for every element in $\Gamma(S_0)$. We will only show the calculations for $x_0 \in \Gamma(S_0)$:

$$th_\Psi^{\gamma_X} \circ \Gamma(s_0)(x_0)(aa) = th_\Psi^{\gamma_X}(x_0)(aa) = 0$$
$$th_\Psi^{\gamma_M} \circ h_1(x_0)(aa) = th_\Psi^{\gamma_M}(m_0)(aa) = 0$$
$$th_\Psi^{\gamma_X} \circ \Gamma(s_0)(x_0)(a) = th_\Psi^{\gamma_X}(x_0)(a) = 1$$
$$th_\Psi^{\gamma_M} \circ h_1(x_0)(a) = th_\Psi^{\gamma_M}(m_0)(a) = 1$$
$$th_\Psi^{\gamma_X} \circ \Gamma(s_0)(x_0)(c) = th_\Psi^{\gamma_X}(x_0)(c) = 0$$
$$th_\Psi^{\gamma_M} \circ h_1(x_0)(c) = th_\Psi^{\gamma_M}(m_0)(c) = 0$$
$$th_\Psi^{\gamma_X} \circ \Gamma(s_0)(x_0)(ac) = th_\Psi^{\gamma_X}(x_0)(ac) = 0$$
$$th_\Psi^{\gamma_M} \circ h_1(x_0)(ac) = th_\Psi^{\gamma_M}(m_0)(ac) = 0$$

As you can see, the only time we actually look at elements of a set is when we construct h, where we look at elements of M, and when we run the tests,

where we look at elements of O^{Ψ}. We can compare this first iteration of testing with running a set of tests $\{a, b, c\} \cdot \Psi$ in a classical sense. The base computation just allows accessing the next states from a given set of states, for every input. So when we do k base computations, it is equivalent to running a set of tests $A^{\leq k} \cdot \Psi$. This inadvertently means that we actually test 'too much', many states will be tested multiple times. However, since we know nothing about the structure of the implementation machine this is hard to prevent. Table 3.1 shows the sequence of base computations for (X, γ_x, x_0). Note that it is not typical that each iteration includes the previous; this is mainly because many states in this example have loops to itself.

i	S_i
0	1
1	$\{x_0, x_1, x_4\}$
2	$\{x_0, x_1, x_2, x_3, x_4\}$
3	$\{x_0, x_1, x_2, x_3, x_4, x_5\}$
4	$\{x_0, x_1, x_2, x_3, x_4, x_5\}$
5	$\{x_0, x_1, x_2, x_3, x_4, x_5\}$
5	$\{x_0, x_1, x_2, x_3, x_4, x_5\}$

Table 3.1: Sequence of base computations for (X, γ_x, x_0)

Now we will test implementation (Y, γ_Y, y_0). This implementation contains a mistake; $\gamma_Y(y_3)(a) = (1, y_3)$ while $\gamma_X(x_3)(a) = (1, x_4)$. This causes a different output for input *baac*. Table 3.2 shows the base computations of each iteration. The algorithm fails when trying to construct a correct h_3. We need the equation $Bh_3 \circ g_2 = \gamma_M \circ h_1$ to hold, and since $g_3(y_3)(a) = (1, y_3)$, $h_1(y_3) = m_3$ and $\gamma_M(m_3)(a) = (1, m_4)$ we have that $h_3(y_3)$ should be equal to m_4. However, we have that $th_{\Psi}^{\gamma_X}(y_3)(c) = 1$ but $th_{\Psi}^{\gamma_M}(m_4)(c) = 0$. Thus a proper h_3 cannot be constructed and testing fails.

i	S_i
0	1
1	$\{y_0, y_1, y_4\}$
2	$\{y_0, y_1, y_2y_3, y_4\}$
3	$\{y_0, y_1, y_2, y_3, y_4\}$

Table 3.2: Sequence of base computations for (Y, γ_Y, y_0)

3.3.2 Labelled Transition Systems

We will now illustrate testing finitely-branching labelled transition systems using our algorithm. Labelled transition systems are coalgebras of the functor $B = \mathcal{P}_\omega(A \times -)$, where $\mathcal{P}_\omega$ is the finite powerset functor. Tests for labelled transition systems are formulas of a variant of Hennessy-Milner logic[5]. Formulas are of the form

$$\phi ::= \langle a \rangle \bigwedge_{j=1..n} \psi_j \qquad\qquad \psi ::= \phi \mid \neg\phi \tag{3.8}$$

We denote the empty conjunction (where $n = 0$) with $\top$. We define our logic $L : \mathsf{Set} \to \mathsf{Set} = A \times \sum_{n\in\mathbb{N}}(2 \times -)^n$ as in [11]. The initial algebra $\alpha : L\Phi \to \Phi$ of this functor is, roughly, the collection of modal formulas generated by grammar 3.8 quotiented by equivalence. A tuple $(a, ((t_1, \phi_1), .., (t_n, \phi_n)) \in L\Phi$, with $a \in A$, $t_i \in 2$ and $\phi_i \in \Phi$ then forms the formula $\alpha(a, ((t_1, \phi_1), .., (t_n, \phi_n)) = \langle a \rangle \bigwedge_{i=1..n} \psi_i$ with $\psi_i = \phi_i$ if $t_i = 0$ and $\psi_i = \neg\phi_i$ if $t_i = 1$.

The one-step semantics $\delta : A \times \sum_{n\in\mathbb{N}}(2 \times 2^-)^n \Rightarrow 2^{\mathcal{P}_\omega(A\times -)}$ is given by:

$$\begin{aligned} &\delta_X(a, ((t_1, \beta_1), .., (t_n, \beta_n)))(S) \\ &\quad = 1 \Leftrightarrow \forall i \in \{0..n\}.\exists (a, x) \in S. \begin{cases} \beta_i(x) = 1 & \text{if } t_i = 0 \\ \beta_i(x) = 0 & \text{if } t_i = 1 \end{cases} \end{aligned} \tag{3.9}$$

Filling the one-step semantics into diagram 2.6 gives us a recursive definition of the semantics map:

$$\begin{aligned} &x \in [\![\langle a \rangle \bigwedge_{i=0..n} \psi_i]\!] \\ &\quad \Leftrightarrow \forall i \in \{0..n\}.\exists (a, y) \in \gamma(x). \begin{cases} y \in [\![\phi_i]\!] & \text{if } \psi_i = \phi_i \\ y \notin [\![\phi_i]\!] & \text{if } \psi_i = \neg\phi_i \end{cases} \end{aligned} \tag{3.10}$$

This amounts to the standard semantics for Hennessy-Milner logic: a formula $\langle a \rangle \bigwedge_{i=0..n} \psi_i$ holds in a state x if x has an a successor y such that all ψ_i hold in y.

Now that we have the semantics of the logic set up, we move to the theory map. The mate of the one-step semantics is given in [11], so we won't recalculate that here:

$$\begin{aligned} &\delta^\flat_\Phi(S)(a, ((t_1, \Psi_1), .., (t_n, \Psi_n)) \\ &\quad = 1 \Leftrightarrow \exists (a, \phi) \in S. \forall i \in \{0..n\}. \begin{cases} \phi \in \Psi_i & \text{if } t_i = 0 \\ \phi \notin \Psi_i & \text{if } t_i = 1 \end{cases} \end{aligned} \tag{3.11}$$

We fill in the mate $\delta^\flat : BQ \Rightarrow QL$ in diagram 2.7, which inductively defines the theory map:

$$2^\alpha \circ th^\gamma = \delta^\flat_\Phi \circ Bth^\gamma \circ \gamma \tag{3.12}$$

Lets start by simplifying the left side:

$$2^\alpha \circ th^\gamma(x) = \{(a, (t_1, \phi_1), .., (t_n, \phi_n)) \in L\Phi \mid \langle a \rangle \bigwedge_{i=o..n} \psi_i \in th^\gamma(x)\}$$

Notice that we can retrieve the theory map by reconstructing the formulas from $QL\Phi$ using α. Now we move on to the right side of equation 3.12 starting with just enumerating the elements of $\gamma(x)$ for clarity:

$$\begin{aligned}
\delta^\flat_\Phi \circ Bth^\gamma \circ \gamma(x) &= \delta^\flat_\Phi \circ Bth^\gamma(\{(a_1, y_1), .., (a_m, y_m)\}) \\
&= \delta^\flat_\Phi(\{(a_1, th^\gamma(y_1)), .., (a_m, th^\gamma(y_m))\}) \\
&= \{(a, ((t_1, \phi_1), .., (t_n, \phi_n))) \in L\Phi \mid \\
&\qquad \exists (a, th^\gamma(y)) \in Bth^\gamma \circ \gamma(x). \forall i = 0..n. \begin{cases} \phi_i \in th^\gamma(x) & \text{if } t_i = 0 \\ \phi_i \notin th^\gamma(x) & \text{if } t_i = 1 \end{cases} \}
\end{aligned}$$

Simplifying this as much as possible, we express the theory map as:

$$\begin{aligned}
&(\langle a \rangle \bigwedge_{i=0..n} \psi_i) \in th^\gamma(x) \Leftrightarrow \\
&\qquad \exists (a, y) \in \gamma(x) \forall \psi_i \begin{cases} \psi_i = \phi_i \Rightarrow \phi_i \in th^\gamma(y) \\ \psi_i = \neg\phi_i \Rightarrow \phi_i \notin th^\gamma(y) \end{cases}
\end{aligned} \tag{3.13}$$

Running a test $\phi = \langle a \rangle \bigwedge_{i=0..n} \psi_i$ in a state $x \in X$ of a labelled transition system $\gamma : X \to BX$ now amounts to checking whether ϕ is contained in the theory map of x, which we will denote by $\phi \in th^\gamma(x)$ or $th^\gamma(x)(\phi) = 1$. This logic is expressive; for a proof, see [11].

Now that we've set up the logical framework we will try to apply it to an example, as we did for Mealy machines. The minimal model we will test against is shown in figure 3.4, and the faulty implementation we will test against is shown in figure 3.5. As always we use the base to compute successors in the implementation, and the theory map to check whether the right formulas hold in the right states. Intuitively, for a labelled transition system, this means that we are guaranteed to be able to access all non-deterministic paths, and that we have some way of checking whether a modal formula holds in a state.

The first step is again coming up with a characterisation for the model. We can separate state m_1 from the other states with the formula $\langle b \rangle \top$, as it is the only state with a b transition. In the same way we can separate

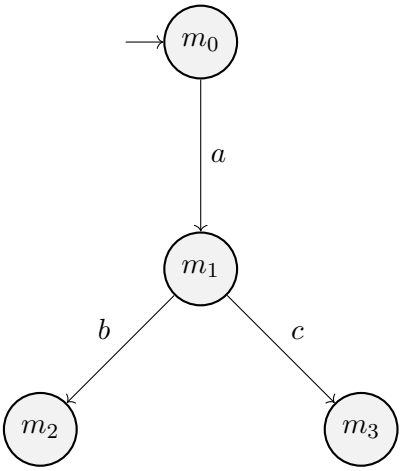


Figure 3.4: Specification LTS (M, γ_M, m_0)

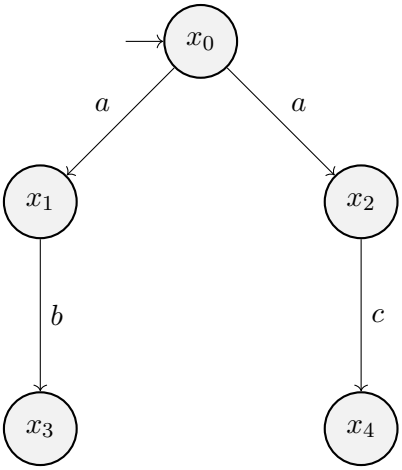


Figure 3.5: An incorrect implementation, (X, γ_X, x_0), of the model (M, γ_M, m_0) defined in figure 3.4.

state m_2 with the formula $\langle c\rangle\top$. The behaviour of the initial state, m_0, can be precisely described by the formula $\langle a\rangle(\langle b\rangle\top) \wedge (\langle c\rangle\top)$, but since we only need to separate it from the other states the formula $\langle a\rangle\top$ suffices. Our characterisation then becomes $\Psi = \{\langle a\rangle\top, \langle b\rangle\top, \langle c\rangle\top\}$.

We can now use the base and our characterisation to begin testing. First, we define $S_0 = \{x_0\}$ and $h_0(x_0) = m_0$. Then we take the base of p_0, where:

$$\begin{aligned}\Gamma(\{p_0\}) &= \{p_1, p_2\}\\ g_0(p_0) &= \{(a, p_1), (a, p_2)\}\end{aligned}$$

We should then get a morphism h from $\text{link}(g_0, h_0)$ such that $Bh \circ g_0 = \gamma_M \circ h_0$. This means we have to define $h_1(x_1) = h_1(x_2) = m_1$. With h_1 constructed we now check if it is correct with respect to the characterisation. We should have that:

$$\begin{aligned}th^{\gamma_X}(x_1) &= th^{\gamma_M}(m_1)\\ th^{\gamma_X}(x_2) &= th^{\gamma_M}(m_1)\end{aligned}$$

This can't hold, since $\langle b\rangle\top$ holds in q_1 but not in p_2 (and $\langle c\rangle\top$ also holds in q_1 but not in p_1). Algorithm 1 fails and we conclude that the implementation in figure 3.5 is incorrect.

Chapter 4

Discussion and Conclusion

4.1 Result and Applicability

The main results of this thesis are the testing algorithm (algorithm 1) and the proofs of its soundness and completeness. We have shown its effectiveness and broad applicability through examples using two very different types of state-based systems, Mealy machines and labelled transition systems, in section 3.3. The work done in this thesis can be seen as a starting point for thinking about testing in a coalgebraic manner. In particular, the way we use separate methods for accessing states and then testing states in a black-box could be a effective way to deal with the problem of having to work with a large variety of state-based systems and the ways those systems receive input.

4.2 Related work

The starting points of this thesis were the generalisation of testing methods for Mealy machines in chapter 2 of [18], and the abstraction of Angluin's L^* algorithm [2] for learning deterministic finite automata to systems described by coalgebra [4]. As in [18], we have distinguished between the part of testing where states of the black-box implementation are accessed. In [18] this is done through the notions of state and transition covers. We use the computation of the base [6] of the functor for this purpose. Our notion of n-completeness (definition 3.7) is an abstraction of the definition in [18].

In [4], a duality between coalgebras and model logic [12, 7] is used to devise the tests that are applied to the system in order to learn its structure. We use this same duality for our tests. The idea to use the base [6] to traverse the black-box system also originates here. Because of this, the framework developed in our thesis should be compatible with the learning algorithm. Since this algorithm contains an equivalence query to an oracle, it could be expanded with our approach, given a bound on the size of the system that

is to be learned, by replacing the oracle query with n-complete testing.

4.3 Future work

Currently, our algorithm only tests for behavioural equivalence, and as such requires the logic used to be expressive. An obvious expansion of the work in this thesis is to adapt the algorithm in such a way that it tests for logical equivalence with respect to the logic used, regardless of the logic's expressivity. This allows testing for other kinds of equivalences, such as trace equivalence, which is more commonly used when testing nondeterministic systems [20], or other relations between state machines, such as the *ioco*-relation [23].

This would require a different method of testing successors in the implementation, which does not require every successor to have a logically equivalent representative in the specification. It might be possible to integrate reachability into the logical framework, rather than relying on the base.

It would also be interesting to try the method on other coalgebraic systems, such as probabilistic systems. Expressive logics for various probabilistic and weighted systems are provided in [10]. Perhaps it is possible to calculate some probability of conformance for these types of systems.

Lastly, in order to automate testing using this method we need a method to build characterisations for coalgebraic specifications. Work on this has been done in [13], where distinguishing modal formulas are built using bisimulation games. It would be interesting to see if we can construct coalgebraic characterisations as defined in this thesis using this method. This would integrate well with the learning algorithm from [4] as it would then be possible to construct counterexamples, rather than an oracle query.

Bibliography

[1] Jirí Adámek, Stefan Milius, and Lawrence S. Moss. On well-founded and recursive coalgebras. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2020.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[3] Steve Awodey. *Category Theory*. Oxford University Press, Inc., USA, 2nd edition, 2010.

[4] Simone Barlocco, Clemens Kupke, and Jurriaan Rot. Coalgebra learning via duality. *CoRR*, abs/1902.05762, 2019.

[5] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[6] Alwin Blok. Interaction, observation and denotation. Master's thesis, 2012.

[7] Marcello M. Bonsangue and Alexander Kurz. Duality for logics of transition systems. In Vladimiro Sassone, editor, *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3441 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 2005.

[8] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[9] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016.

[10] Bart Jacobs and Ana Sokolova. Exemplaric expressivity of modal logics. *J. Log. Comput.*, 20(5):1041–1068, 2010.

[11] Bartek Klin. Coalgebraic modal logic beyond sets. *Electronic Notes in Theoretical Computer Science*, 173:177–201, 2007. Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII).

[12] Clemens Kupke, Alexander Kurz, and Dirk Pattinson. Algebraic semantics for coalgebraic logics. In Jirí Adámek and Stefan Milius, editors, *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, CMCS 2004, Barcelona, Spain, March 27-29, 2004*, volume 106 of *Electronic Notes in Theoretical Computer Science*, pages 219–241. Elsevier, 2004.

[13] Barbara König, Christina Mika-Michalski, and Lutz Schröder. Explaining non-bisimilarity in a coalgebraic approach: Games and distinguishing formulas. *Lecture Notes in Computer Science*, page 133–154, 2020.

[14] Stephen Lack and Pawel Sobocinski. Adhesive categories. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2004.

[15] Stephen Lack and Pawel Sobocinski. Toposes are adhesive. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2006.

[16] David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. Computers*, 43(3):306–320, 1994.

[17] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Software Eng.*, 20(2):149–162, 1994.

[18] Joshua Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. PhD thesis, Radboud Universiteit Nijmegen, 2019.

[19] E.F. Moore. Gedanken-experiments on sequential machines. *Sequential Machines,Automata Studies, Annals of Mathematical Studies, no.34*, 1956.

[20] Alexandre Petrenko and Nina Yevtushenko. Adaptive testing of nondeterministic systems with FSM. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*, pages 224–228. IEEE Computer Society, 2014.

[21] Jan Rutten. *The Method of Coalgebra: exercises in coinduction*. Centrum Wiskunde Informatica, Amsterdam, The Netherlands, 2019.

[22] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[23] Petra van den Bos, Ramon Janssen, and Joshua Moerman. n-complete test suites for IOCO. *Softw. Qual. J.*, 27(2):563–588, 2019.

[24] M.P. Vasilevskii. Failure diagnosis of automata. *Cybernetics and Systems Analysis*, 9(4):653–665, 1956.