Nominal Techniques and Black Box Testing for Automata Learning

Joshua Moerman

Radboud Universiteit

Work in the thesis has been carried out under the auspices of the research school *IPA* (Institute for Programming research and Algorithmics)

Printed by Gildeprint, Enschede

Typeset using ConTEXt MKIV

ISBN: 978–94–632–3696–6

IPA Dissertation series: 2019-06

www.joshuamoerman.nl

Nominal Techniques and Black Box Testing for Automata Learning

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen

op
maandag 1 juli 2019
om
16:30 uur precies

door

Joshua Samuel Moerman
geboren op 1 oktober 1991
te Utrecht

Promotoren:
- prof. dr. F.W. Vaandrager
- prof. dr. A. Silva (University College London, Verenigd Koninkrijk)

Copromotor:
- dr. S.A. Terwijn

Leden manuscriptcommissie:
- prof. dr. B.P.F. Jacobs
- prof. dr. A.R. Cavalli (Télécom SudParis, Frankrijk)
- prof. dr. F. Howar (Technische Universität, Dortmund, Duitsland)
- prof. dr. S. Lasota (Uniwesytet Warszawkski, Polen)
- dr. D. Petrişan (Université Paris Diderot, Frankrijk)

Paranimfen:
- Alexis Linard
- Tim Steenvoorden

Samenvatting

Het *leren van automaten* speelt een steeds grotere rol bij de verificatie van software. Tijdens het leren, verkent een leeralgoritme het gedrag van software. Dit gaat in principe volledig automatisch, en het algoritme pakt vanzelf interessante eigenschappen op van de software. Het is hiermee mogelijk een redelijk precies model te maken van de werking van het stukje software dat we onder de loep nemen. Fouten en onverwacht gedrag van software kunnen hiermee worden blootgelegd.

In dit proefschrift kijken we in eerste instantie naar technieken voor *testgeneratie*. Deze zijn nodig om het leeralgoritme een handje te helpen. Na het automatisch verkennen van gedrag, formuleert het leeralgoritme namelijk een hypothese die de software nog niet goed genoeg modelleert. Om de hypothese te verfijnen en verder te leren, hebben we tests nodig. *Efficiëntie* staat hierbij centraal: we willen zo min mogelijk testen, want dat kost tijd. Aan de andere kant moeten we wel *volledig testen:* als er een discrepantie is tussen het geleerde model en de software, dan willen we die met een test kunnen aanwijzen.

In de eerste paar hoofdstukken laten we zien hoe testen van automaten te werk gaat. We geven een theoretisch kader om verschillende, bestaande n-*volledige testgeneratiemethodes* te vergelijken. Op grond hiervan beschrijven we een nieuw, efficiënt algoritme. Dit nieuwe algoritme staat centraal bij een industriële casus waarin we een model van complexe printer-software van Océ leren. We laten ook zien hoe een van de deelproblemen – het *onderscheiden van toestanden* met zo kort mogelijke invoer – efficiënt kan worden opgelost.

Het tweede thema in dit proefschrift is de theorie van formele talen en automaten met *oneindige alfabetten*. Ook dit is zinnig voor het leren van automaten. Software, en in het bijzonder internet-communicatie-protocollen, maken namelijk vaak gebruik van „identifiers" om bijvoorbeeld verschillende gebruikers te onderscheiden. Het liefst nemen we oneindig veel van zulke identifiers aan, aangezien we niet weten hoeveel er nodig zijn voor het leren van de automaat.

We laten zien hoe we de leeralgoritmes gemakkelijk kunnen veralgemeniseren naar oneindige alfabetten door gebruik te maken van *nominale verzamelingen*. In het bijzonder kunnen we hiermee registerautomaten leren. Vervolgens werken we de theorie van nominale automaten verder uit. We laten zien hoe je deze structuren efficiënt kan implementeren. En we geven een speciale klasse van nominale automaten die een veel kleinere representatie hebben. Dit zou gebruikt kunnen worden om zulke automaten sneller te leren.

Summary

Automata learning plays a more and more prominent role in the field of software verification. Learning algorithms are able to automatically explore the behaviour of software. By revealing interesting properties of the software, these algorithms can create models of the, otherwise unknown, software. These learned models can, in turn, be inspected and analysed, which often leads to finding bugs and inconsistencies in the software.

An important tool which we need when learning software is *test generation.* This is the topic of the first part of this thesis. After the learning algorithm has learned a model and constructed a hypothesis, test generation methods are used to validate this hypothesis. *Efficiency* is key: we want to test as little as possible, as testing may take valuable time. However, our tests have to be *complete:* if the hypothesis fails to model the software well, we better have a test which shows this discrepancy.

The first few chapters explain black box testing of automata. We present a theoretical framework in which we can compare existing n*-complete test generation methods.* From this comparison, we are able to define a new, efficient algorithm. In an industrial case study on embedded printer software, we show that this new algorithm works well for finding counterexamples for the hypothesis. Besides the test generation, we show that one of the subproblems – finding the shortest sequences to separate states – can be solved very efficiently.

The second part of this thesis is on the theory of formal languages and automata with *infinite alphabets*. This, too, is discussed in the context of automata learning. Many pieces of software make use of identifiers or sequence numbers. These are used, for example, in order to distinguish different users or messages. Ideally, we would like to model such systems with infinitely many identifiers, as we do not know beforehand how many of them will be used.

Using the theory of *nominal sets*, we show that learning algorithms can easily be generalised to automata with infinite alphabets. In particular, this shows that we can learn register automata. Furthermore, we deepen the theory of nominal sets. First, we show that, in a special case, these sets can be implemented in an efficient way. Second, we give a subclass of nominal automata which allow for a much smaller representation. This could be useful for learning such automata more quickly.

Acknowledgements

Foremost, I would like to thank my supervisors. Having three of them ensured that there were always enough ideas to work on, theory to understand, papers to review, seminars to attend, and chats to have. Frits, thank you for being a very motivating supervisor, pushing creativity, and being only a few meters away. It started with a small puzzle (trying a certain test algorithm to help with a case study), which was a great, hands-on start of my Ph.D.. You introduced me to the field of model learning in a way that showcases both the theoretical and practical aspects.

Alexandra, thanks for introducing me to abstract reasoning about state machines, the coalgebraic way. Although not directly shown in this thesis, this way of thinking has helped and you pushed me to pursuit clear reasoning. Besides the theoretical things I've learned, you have also taught me many personal lessons inside and outside of academia; thanks for inviting me to London, Caribbean islands, hidden cocktail clubs, and the best food. And thanks for leaving me with Daniela and Matteo, who introduced me to nominal techniques, while you were on sabbatical.

Bas, thanks for broadening my understanding of the topics touched upon in this thesis. Unfortunately, we have no papers together, but the connections you showed to logic, computational learning, and computability theory have influenced the thesis nevertheless. I am grateful for the many nice chats we had.

I would like to thank the members of the manuscript committee, Bart, Ana, Falk, Sławek, and Daniela. Reading a thesis is undoubtedly a lot of work, so thank you for the effort and feedback you have given me. Thanks, also, to the additional members coming to Nijmegen to oppose during the defence, Jan Friso, Jorge, and Paul.

On the first floor of the Mercator building, I had the pleasure of spending four years with fun office mates. Michele, thanks for introducing me to the Ph.D. life, by always joking around. Hopefully, we can play a game of Briscola again. Alexis, many thanks for all the tasty *proeverijen,* whether it was beers, wines, poffertjes, kroketten, or anything else. Your French influences will be missed. Niels, thanks for the abstract nonsense and bashing on politics.

Next to our office, was the office with Tim, with whom I had the pleasure of working from various coffee houses in Nijmegen. Further down the corridor, there was the office of Paul and Rick. Paul, thanks for being the kindest colleague I've had and for inviting us to your musical endeavours. Rick, thanks for the algorithmic sparring, we had a great collaboration. Was there a more iconic duo on our floor? A good contender would be Petra and Ramon. Thanks for the fun we had with ioco, together with Jan and Mariëlle. Nils, thanks for steering me towards probabilistic

things and opening a door to Aachen. I am also very grateful to Jurriaan for bringing back some coalgebra and category theory to our floor, and hosting me in London. My other co-authors, Wouter, David, Bartek, Michał, and David, also deserve many credits for all the interesting discussion we had. Harco, thanks for the technical support. Special thanks go to Ingrid, for helping with the often-overlooked, but important, administrative matters.

Doing a Ph.D. would not be complete without a good amount of playing kicker, having borrels, and eating cakes at the iCIS institute. Thanks to all of you, Markus, Bram, Marc, Sam, Bas, Joost, Dan, Giso, Baris, Simone, Aleks, Manxia, Leon, Jacopo, Gabriel, Michael, Paulus, Marcos, Bas, and Henning.[1]

Thanks to the people I have met across the channel (which hopefully will remain part of the EU): Benni, Nath, Kareem, Rueben, Louis, Borja, Fred, Tobias, Paul, Gerco, and Carsten, for the theoretical adventure, but also for joining me to *Phonox* and other parties in London. I am especially thankful to Matteo and Emanuela for hosting me many times and for Hillary and Justin for accommodating me for three months each.

I had a lot of fun at the IPA events. I'm very thankful to Tim and Loek for organising these events. Special thanks to Nico and Priyanka for organising a Halloween social event with me. Also thanks to all the participants in the IPA events, you made it a lot of fun! My gratitude extends to all the people I have met at summer schools and conferences. I had a lot of fun learning about different cultures, languages, and different ways of doing research. Hope we meet again!

Besides all the fun research, I had a great time with my friends and family. We went to nice parties, had excellent dinners, and much more; thanks, Nick, Edo, Gabe, Saskia, Stijn, Sandra, Geert, Marco, Carmen, and Wesley. Thanks to Marlon, Hannah, Wouter, Dennis, Christiaan, and others from #RU for borrels, bouldering, and jams. Thanks to Ragnar, Josse, Julian, Jeroen, Vincent, and others from the BAPC for algorithmic fun.

Thanks to my parents, Kees and Irene, and my brother, David, and his wife, Germa, for their love and support. My gratitude extends to my family in law, Ine, Wim, Jolien and Jesse. My final words of praise go to Tessa, my wife, I am very happy to have you on my side. You inspire me in many ways, and I enjoy doing all the fun stuff we do. Thank you a lot.

[1] In no particular order. These lists are randomised.

Contents

Chapter 1
Introduction

When I was younger, I often learned how to play with new toys by messing about with them, by pressing buttons at random, observing their behaviour, pressing more buttons, and so on. Only resorting to the manual – or asking "experts" – to confirm my beliefs on how the toys work. Now that I am older, I do mostly the same with new devices, new tools, and new software. However, now I know that this is an established computer science technique, called *model learning*.

Model learning[2] is an automated technique to construct a state-based model – often a type of *automaton* – from a black box system. The goal of this technique can be manifold: it can be used to reverse-engineer a system, to find bugs in it, to verify properties of the system, or to understand the system in one way or another. It is *not* just random testing: the information learned during the interaction with the system is actively used to guide following interactions. Additionally, the information learned can be inspected and analysed.

This thesis is about model learning and related techniques. In the first part, I present results concerning *black box testing* of automata. Testing is a crucial part in learning software behaviour and often remains a bottleneck in applications of model learning. In the second part, I show how *nominal techniques* can be used to learn automata over structured infinite alphabets. The study on nominal automata was directly motivated by work on learning network protocols which rely on identifiers or sequence numbers.

But before we get ahead of ourselves, we should first understand what we mean by learning, as learning means very different things to different people. In educational science, learning may involve concepts such as teaching, blended learning, and interdisciplinarity. Data scientists may think of data compression, feature extraction, and neural networks. In this thesis we are mostly concerned with software verification. But even in the field of verification several types of learning are relevant.

1 *Model Learning*

In the context of software verification, we often look at stateful computations with inputs and outputs. For this reason, it makes sense to look at *words*, or *traces*. For an alphabet Σ, we denote the set of words by Σ^*.

[2] There are many names for the type of learning, such as *active automata learning*. The generic name "model learning" is chosen as a counterpoint to model checking.

The learning problem is defined as follows. There is some fixed, but unknown, language $\mathcal{L} \subseteq \Sigma^*$. This language may define the behaviour of a software component, a property in model checking, a set of traces from a protocol, etc. We wish to infer a description of $\mathcal{L}$ after only having observed a small part of this language. For example, we may have seen hundred words belonging to the language and a few which do not belong to the language. Then concluding with a good description of $\mathcal{L}$ is difficult, as we are missing information about the infinitely many words we have not observed.

Such a learning problem can be stated and solved in a variety of ways. In the applications we do in our research group, we often try to infer a model of a software component. (Chapter 3 describes such an application.) In these cases, a learning algorithm can interact with the software. So it makes sense to study a learning paradigm which allows for *queries*, and not just a data set of samples.

A typical query learning framework was established by Angluin (1987). In her framework, the learning algorithm may pose two types of queries to a *teacher*, or *oracle*:

Membership queries (MQ) The learner poses such a query by providing a word $w \in \Sigma^*$ to the teacher. The teacher will then reply whether $w \in \mathcal{L}$ or not. This type of query is often generalised to more general output, in these cases we consider $\mathcal{L} : \Sigma^* \to O$ and the teacher replies with $\mathcal{L}(w)$. In some papers, such a query is then called an *output query*.

Equivalence queries (EQ) The learner can provide a hypothesised description H of $\mathcal{L}$ to the teacher. If the hypothesis is correct, the teacher replies with `yes`. If, however, the hypothesis is incorrect, the teacher replies with `no` together with a counterexample, i.e., a word which is in $\mathcal{L}$ but not in the hypothesis or vice versa.

By posing many such queries, the learner algorithm is supposed to converge to a correct model. This type of learning is hence called *exact learning*. Angluin (1987) showed that one can do this efficiently for deterministic finite automata (DFAs), when $\mathcal{L}$ is in the class of regular languages.

It should be clear why this is called *query learning* or *active learning*. The learning algorithm initiates interaction with the teacher by posing queries, it may construct its own data points and ask for their corresponding label. Active learning is in contrast to *passive learning* where all observations are given to the algorithm up front.

Another paradigm which is relevant for our type of applications is *PAC-learning with membership queries*. Here, the algorithm can again use MQs as before, but the EQs are replaced by random sampling. So the allowed query is:

Random sample queries (EX) If the learner poses this query (there are no parameters), the teacher responds with a random word w together with its label, i.e., whether $w \in \mathcal{L}$ or not. (Here, *random* means that the words are sampled by some probability distribution known to the teacher.)

Instead of requiring that the learner exactly learns the model, we only require the following. The learner should *probably* return a model which is *approximate* to the target. This gives the name *probably approximately correct* (PAC). Note that there are two uncertainties: the probable and the approximate part. Both parts are bounded by parameters, so one can determine the confidence.

As with many problems in computer science, we are also interested in the *efficiency* of learning algorithms. Instead of measuring time or space, we analyse the number of queries posed by an algorithm. Efficiency often means that we require a polynomial number of queries. But polynomial in what? The learner has no input, other than the access to a teacher. We ask the algorithms to be polynomial in the *size of the target* (i.e., the size of the description which has yet to be learned). In the case of PAC learning we also require it to be polynomial in the two parameters for confidence.

Deterministic automata can be efficiently learned in the PAC model. In fact, any efficient exact learning algorithm with MQs and EQs can be transformed into an efficient PAC algorithm with MQs (see Kearns & Vazirani, 1994, exercise 8.1). For this reason, we mostly focus on the former type of learning in this thesis. The transformation from exact learning to PAC learning is implemented by simply *testing* the hypothesis with random samples. This can be postponed until we actually implement a learning algorithm and apply it.

When using only EQs, only MQs, or only EXs, then there are hardness results for exact learning of DFAs. So the combinations MQs + EQs (for exact learning) and MQs + EXs (for PAC learning) have been carefully picked, they provide a minimal basis for efficient learning. See the book of Kearns and Vazirani (1994) for such hardness results and more information on PAC learning.

So far, all the queries are assumed to be *just there*. Somehow, these are existing procedures which we can invoke with $\texttt{MQ}(w)$, $\texttt{EQ}(H)$, or $\texttt{EX}()$. This is a useful abstraction when designing a learning algorithm. One can analyse the complexity (in terms of number of queries) independently of how these queries are resolved. Nevertheless, at some point in time one has to implement them. In our case of learning software behaviour, membership queries are easily implemented: simply provide the word w to a running instance of the software and observe the output.[3] Equivalence queries, however, are in general not doable. Even if we have the (machine) code, it is often way too complicated to check equivalence. That is why we resort to testing with EX queries. The EX query from PAC learning normally assumes a fixed, unknown probability distribution on words. In our case, we choose and implement a distribution to test against. This cuts both ways: On the one hand, it allows us to only test behaviour we really care about, on the other hand the results are only as good as our choice of distribution. We deviate even further from the PAC-model as we sometimes change

[3] In reality, it is a bit harder than this. There are plentiful of challenges to solve, such as timing, choosing your alphabet, choosing the kind of observations to make, and being able to reliably reset the software.

our distribution while learning. Yet, as applications show, this is a useful way of learning software behaviour.

2 *Applications of Model Learning*

Since this thesis contains only one real-world application of learning in Chapter 3, it is good to mention a few others. Although we remain in the context of learning software behaviour, the applications are quite different from each other. This is by no means a complete list.

Bug finding in protocols. A prominent example is by Fiterău-Broștean, et al. (2016). They learn models of TCP implementations – both clients and server sides. Interestingly, they found bugs in the (closed source) Windows implementation. Later, Fiterău-Broștean and Howar (2017) also found a bug in the sliding window of the Linux implementation of TCP. Other protocols have been learned as well, such as the MQTT protocol by Tappler, et al. (2017), TLS by de Ruiter and Poll (2015), and SSH by Fiterău-Broștean, et al. (2017). Many of these applications reveal bugs by learning a model and consequently apply model checking. The combination of learning and model checking was first described by Peled, et al. (2002).

Bug finding in smart cards. Aarts, et al. (2013) learn the software on smart cards of several Dutch and German banks. These cards use the EMV protocol, which is run on the card itself. So this is an example of a real black box system, where no other monitoring is possible and no code is available. No vulnerabilities were found, although each card had a slightly different state machine. The e.dentifier, a card reader implementing a challenge-response protocol, has been learned by Chalupar, et al. (2014). They built a Lego machine which could automatically press buttons and the researchers found a security flaw in this card reader.

Regression testing. Hungar, et al. (2003) describe the potential of automata learning in regression testing. The aim is not to find bugs, but to monitor the development process of a system. By considering the differences between models at different stages, one can generate regressions tests.

Refactoring legacy software. Model learning can also be used in order to verify refactored software. Schuts, et al. (2016) have applied this at a project within Philips. They learn both an old version and a new version of the same component. By comparing the learned models, some differences could be seen. This gave developers opportunities to solve problems before replacing the old component by the new one.

3 Research challenges

In this thesis, we will mostly see learning of deterministic automata or Mealy machines. Although this is limited, as many pieces of software require richer models, it has been successfully applied in the above examples. The limitations include the following.

- The system behaves deterministically.
- One can reliably reset the system.
- The system can be modelled with a finite state space. This also means that the model does not incorporate time or data.
- The input alphabet is finite.
- One knows when the target is reached.

Research challenge 1: Approximating equivalence queries. Having confidence in a learned model is difficult. We have PAC guarantees (as discussed before), but sometimes we may want to draw other conclusions. For example, we may require the hypothesis to be correct, provided that the real system is implemented with a certain number of states. Efficiency is important here: We want to obtain those guarantees fast and we want to quickly find counterexamples when the hypothesis is wrong. Test generation methods is the topic of the first part in this thesis. We will review existing algorithms and discuss new algorithms for test generation.

Research challenge 2: Generalisation to infinite alphabets. Automata over infinite alphabets are very useful for modelling protocols which involve identifiers or time-stamps. Not only is the alphabet infinite in these cases, the state space is as well, since the values have to be remembered. In the second part of this thesis, we will see how nominal techniques can be used to tackle this challenge.

Being able to learn automata over an infinite alphabet is not new. It has been tackled, for instance, by Howar, et al. (2012), Bollig, et al. (2013) and in the theses of Aarts (2014), Cassel (2015), and Fiterău-Broștean (2018). In the first thesis, the problem is solved by considering abstractions, which reduces the alphabet to a finite one. These abstractions are automatically refined when a counterexample is presented to the algorithms. Fiterău-Broștean (2018) extends this approach to cope with "fresh values", crucial for protocols such as TCP. In the thesis by Cassel (2015), another approach is taken. The queries are changed to tree queries. The approach in my thesis will be based on symmetries, which gives yet another perspective into the problem of learning such automata.

4 Black Box Testing

An important step in automata learning is equivalence checking. Normally, this is abstracted away and done by an oracle, but we intend to implement such an oracle

ourselves for our applications. Concretely, the problem we need to solve is that of *conformance checking*[4] as it was first described by Moore (1956).

The problem is as follows: Given the description of a finite state machine and a black box system, does the system behave exactly as the description? We wish to determine this by running experiments on the system (as it is black box). It should be clear that this is a hopelessly difficult task, as an error can be hidden arbitrarily deep in the system. That is why we often assume some knowledge of the system. In this thesis we often assume a *bound on the number of states* of the system. Under these conditions, Moore (1956) already solved the problem. Unfortunately, his experiment is exponential in size, or in his own words: "fantastically large."

Years later, Chow (1978) and Vasilevskii (1973) independently designed efficient experiments. In particular, the set of experiments is polynomial in the number of states. These techniques will be discussed in detail in Chapter 2. More background and other related problems, as well as their complexity results, are well exposed in a survey of Lee and Yannakakis (1994).

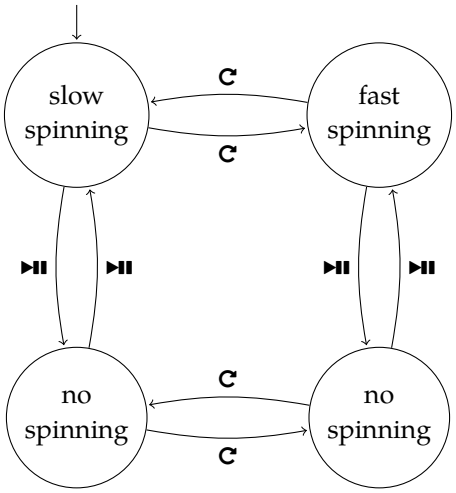


Figure 1.1 Behaviour of a record player modelled as a finite state machine.

To give an example of conformance checking, we model a record player as a finite state machine. We will not model the audible output – that would depend not only on the device, but also the record one chooses to play[5]. Instead, the only observation we can make is looking how fast the turntable spins. The device has two buttons: a

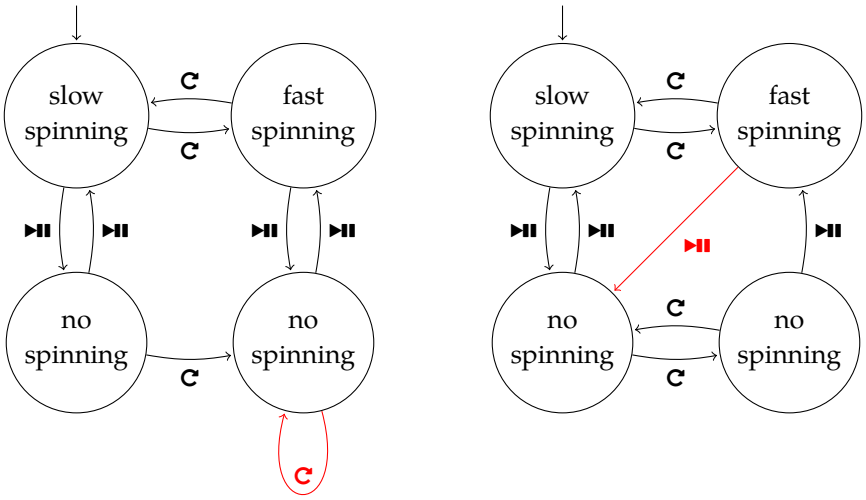
[4] Also known as *machine verification* or *fault detection*.

[5] In particular, we have to add time to the model as one side of a record only lasts for roughly 25 minutes. Unless we take a record with sound on the locked groove such as the *Sgt. Pepper's Lonely Hearts Club Band* album by *The Beatles*.

start-stop button (⏯) and a speed button (🔁) which toggles between $33\frac{1}{3}$ rpm and 45 rpm. When turned on, the system starts playing immediately at $33\frac{1}{3}$ rpm – this is useful for DJing. The intended behaviour of the record player has four states as depicted in Figure 1.1.

Let us consider some faults which could be present in an implementation with four states. In Figure 1.2, two flawed record players are given. In the first (Figure 1.2a), the sequence ⏯🔁🔁 leads us to the wrong state. However, this is not immediately observable, the turntable is in a non-spinning state as it should be. The fault is only visible when we press ⏯ once more: now the turntable is spinning fast instead of slow. The sequence ⏯🔁🔁⏯ is a *counterexample*. In the second example (Figure 1.2b), the fault is again not immediately obvious: after pressing 🔁⏯ we are in the wrong state as observed by pressing ⏯. Here, the counterexample is 🔁⏯⏯.

When a model of the implementation is given, it is not hard to find counterexamples. However, in a black box setting we do not have such a model. In order test whether a black box system is equivalent to a model, we somehow need to test all possible counterexamples. In this example, a test suite should include sequences such as ⏯🔁🔁⏯ and 🔁⏯⏯.



(a) (b)

Figure 1.2 Two faulty record players.

5 *Nominal Techniques*

In the second part of this thesis, I will present results related to *nominal automata*. Usually, nominal techniques are introduced in order to solve problems involving name binding in topics like lambda calculus. However, we use them in automata

theory, specifically to model *register automata*. These are automata which have an infinite alphabet, often thought of as input actions with *data*. The control flow of the automaton may actually depend on the data. However, the data cannot be used in an arbitrary way as this would lead to many decision problems, such as emptiness and equivalence, being undecidable.[6] A principal concept in nominal techniques is that of *symmetries*.

To motivate the use of symmetries, we will look at an example of a register automaton. In the following automaton we model a (not-so-realistic) login system for a single person. The alphabet consists of the following actions:

`sign-up`(p)	`logout`()
`login`(p)	`view`()

The `sign-up` action allows one to set a password p. This can only be done when the system is initialised. The `login` and `logout` actions speak for themselves and the `view` action allows one to see the secret data (we abstract away from what the user actually gets to see here). A simple automaton with roughly this behaviour is given in Figure 1.3. We will only informally discuss its semantics for now.

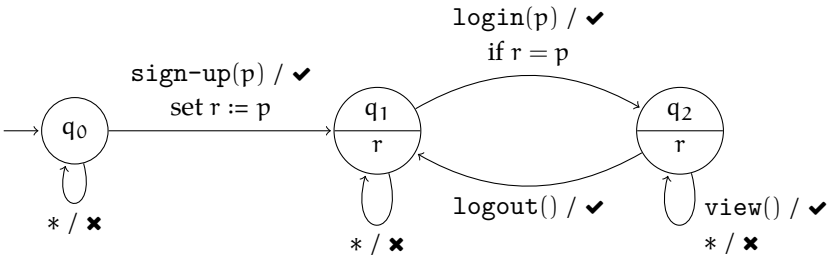


Figure 1.3 A simple register automaton. The symbol ∗ denotes any input otherwise not specified. The r in states q_1 and q_2 is a register.

To model the behaviour, we want the domain of passwords to be infinite. After all, one should allow arbitrarily long passwords to be secure. This means that a register automaton is actually an automaton over an infinite alphabet.

Common algorithms for automata, such as learning, will not work with an infinite alphabet. Any loop which iterates over the alphabet will diverge. In order to cope with this, we will use the *symmetries* present in the alphabet.

Let us continue with the example and look at its symmetries. If a person signs up with a password "`hello`" and consequently logins with "`hello`", then this is not distinguishable from a person signing up and logging in with "`bye`". This is an example of symmetry: the values "`hello`" and "`bye`" can be permuted, or interchanged. Note, however, that the trace `sign-up`(`hello`) `login`(`bye`) is different from the two before:

[6] The class of automata with arbitrary data operations is sometimes called *extended finite state machines*.

no permutation of "hello" and "bye" will bring us to a logged-in state with that trace. So we see that, despite the symmetry, we cannot simply identify the value "hello" and "bye". For this reason, we keep the alphabet infinite and explicitly mention its symmetries.

Using symmetries in automata theory is not a new idea. In the context of model checking, the first to use symmetries were Emerson and Sistla (1996) and Ip and Dill (1996). But only Ip and Dill (1996) used it to deal with infinite data domains. For automata learning with infinite domains, symmetries were used by Sakamoto (1997). He devised an L* learning algorithm for register automata, much like the one presented in Chapter 5. The symmetries are crucial to reduce the problem to a finite alphabet and use the regular L* algorithm. (Chapter 5 shows how to do it with more general symmetries.) Around the same time Ferrari, et al. (2005) worked on automata theoretic algorithms for the π-calculus. Their approach was based on the same symmetries and they developed a theory of *named sets* to implement their algorithms. Named sets are equivalent to nominal sets. However, nominal sets are defined in a more elementary way. The nominal sets we will soon see are introduced by Gabbay and Pitts (2002) to solve certain problems in name binding in abstract syntaxes. Although this is not really related to automata theory, it was picked up by Bojańczyk, et al. (2014), who provide an equivalence between register automata and nominal automata. (This equivalence is exposed in more detail in the book of Bojańczyk, 2018.) Additionally, they generalise the work on nominal sets to other symmetries.

The symmetries we encounter in this thesis are listed below, but other symmetries can be found in the literature. The symmetry directly corresponds to the data values (and operations) used in an automaton. The data values are often called *atoms*.

– The *equality symmetry*. Here the domain can be any countably infinite set. We can take, for example, the set of strings we used before as the domain from which we take passwords. No further structure is used on this domain, meaning that any value is just as good as any other. The symmetries therefore consist of all bijections on this domain.

– The *total order symmetry*. In this case, we take a countable infinite set with a dense total order. Typically, this means we use the rational numbers, $\mathbb{Q}$, as data values and symmetries which respect the ordering.

5.1 What is a nominal set?

So what exactly is a nominal set? I will not define it here and leave the formalities to the corresponding chapters. It suffices, for now, to think of nominal sets as abstract sets (often infinite) on which a group of symmetries acts. This action makes it possible to interpret the symmetries of the data values in the abstract set. For automata, this

allows us to talk about symmetries on the state space, the set of transitions, and the alphabet.

In order to implement these sets algorithmically, we impose two finiteness requirements. Both properties can be expressed using only the group action.

– Each element is *finitely supported*. A way to think of this requirement is that each element is "constructed" out of finitely many data values.

– The set is *orbit-finite*. This means that we can choose finitely many elements such that any other element is a permuted version of one of those elements.

If we wish to model the automaton from Figure 1.3 as a nominal automaton, then we can simply define the state space as

$$Q = \{q_0\} \cup \{q_{1,a} \mid a \in \mathbb{A}\} \cup \{q_{2,a} \mid a \in \mathbb{A}\},$$

where $\mathbb{A}$ is the set of atoms. In this example, $\mathbb{A}$ is the set of all possible passwords. The set Q is infinite, but satisfies the two finiteness requirements.

The upshot of doing this is that the set Q (and transition structure) corresponds directly to the semantics of the automaton. We do not have to *encode* how values relate or how they interact. Instead, the set (and transition structure) defines all we need to know. Algorithms, such as reachability, minimisation, and learning, can be run on such automata, despite the sets being infinite. These algorithms can be implemented rather easily by using a libraries such as Nλ, Lois, or Ons from Chapter 6. These libraries implement a data structure for nominal sets, and provide ways to iterate over such (infinite) sets.

One has to be careful as not all results from automata theory transfer to nominal automata. A notable example is the powerset construction, which converts a non-deterministic automaton into a deterministic one. The problem here is that the powerset of a set is generally *not* orbit-finite and so the finiteness requirement is not met. Consequently, languages accepted by nominal DFAs are *not* closed under Kleene star, or even concatenation.

6 Contributions

This thesis is split into two parts. Part 1 contains material about black box testing, while Part 2 is about nominal techniques. The chapters can be read in isolation. However, the chapters do get more technical and mathematical – especially in Part 2.

Detailed discussion on related work and future directions of research are presented in each chapter.

Chapter 2: FSM-based test methods. This chapter introduces test generation methods which can be used for learning or conformance testing. The methods are presented

in a uniform way, which allows to give a single proof of completeness for all these methods. Moreover, the uniform presentation gives room to develop new test generation methods. The main contributions are:

- Uniform description of known methods: Theorem 26 (p. 35)
- A new proof of completeness: Section 5 (p. 36)
- New algorithm (*hybrid ADS*) and its implementation: Section 3.2 (p. 34)

Chapter 3: Applying automata learning to embedded control software. In this chapter we will apply model learning to an industrial case study. It is a unique benchmark as it is much bigger than any of the applications seen before (3410 states and 77 inputs). This makes it challenging to learn a model and the main obstacle is finding counterexamples. The main contribution is:

- Application of the *hybrid ADS* algorithm: Section 2.2 (p. 49)
- Succesfully learn a large-scale system: Section 2.3 (p. 51)

This is based on the following publication:
Smeenk, W., Moerman, J., Vaandrager, F. W., & Jansen, D. N. (2015). Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM, Proceedings*. Springer. doi:10.1007/978-3-319-25423-4_5.

Chapter 4: Minimal separating sequences for all pairs of states. Continuing on test generation methods, this chapter presents an efficient algorithm to construct separating sequences. Not only is the algorithm efficient – it runs in $\mathcal{O}(n \log n)$ time – it also constructs minimal length sequences. The algorithm is inspired by a minimisation algorithm by Hopcroft (1971), but extending it to construct witnesses is non-trivial. The main contributions are:

- Efficient algorithm for separating sequences: Algorithms 4.2 & 4.4 (p. 66 & 68)
- Applications to black box testing: Section 4 (p. 70)
- Implementation: Section 5 (p. 71)

This is based on the following publication:
Smetsers, R., Moerman, J., & Jansen, D. N. (2016). Minimal Separating Sequences for All Pairs of States. In *Language and Automata Theory and Applications - 10th International Conference, LATA, Proceedings*. Springer. doi:10.1007/978-3-319-30000-9_14.

Chapter 5: Learning nominal automata. In this chapter, we show how to learn automata over infinite alphabets. We do this by translating the L^* algorithm directly to a nominal version, νL^*. The correctness proofs mimic the original proofs by Angluin

(1987). Since our new algorithm is close to the original, we are able to translate variants of the L* algorithm as well. In particular, we provide a learning algorithm for nominal non-deterministic automata. The main contributions are:

- L*-algorithm for nominal automata: Section 3 (p. 86)
- Its correctness and complexity: Theorem 7 & Corollary 11 (p. 89 & 93)
- Generalisation to non-deterministic automata: Section 4.2 (p. 96)
- Implementation in Nλ: Section 5.2 (p. 103)

This is based on the following publication:
Moerman, J., Sammartino, M., Silva, A., Klin, B., & Szynwelski, M. (2017). Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*. ACM. doi:10.1145/3009837.3009879.

Chapter 6: Fast computations on ordered nominal sets. In this chapter, we provide a library to compute with nominal sets. We restrict our attention to nominal sets over the total order symmetry. This symmetry allows for a rather easy characterisation of orbits, and hence an easy implementation. We experimentally show that it is competitive with existing tools, which are based on SMT solvers. The main contributions are:

- Characterisation theorem of orbits: Table 6.1 (p. 118)
- Complexity results: Theorems 18 & 21 (p. 119 and 123)
- Implementation: Section 3 (p. 118)

This is based on the following publication:
Venhoek, D., Moerman, J., & Rot, J. (2018). Fast Computations on Ordered Nominal Sets. In *Theoretical Aspects of Computing - ICTAC - 15th International Colloquium, Proceedings*. Springer. doi:10.1007/978-3-030-02508-3_26.

Chapter 7: Separation and Renaming in Nominal Sets. We investigate how to reduce the size of certain nominal automata. This is based on the observation that some languages (with outputs) are not just invariant under symmetries, but invariant under arbitrary *transformations*, or *renamings*. We define a new type of automaton, the *separated nominal automaton*, and show that they exactly accept those languages which are closed under renamings. All of this is shown by using a theoretical framework: we establish a strong relationship between nominal sets on one hand, and nominal renaming sets on the other. The main contributions are:

- Adjunction between nominal sets and renaming sets: Theorem 16 (p. 138)
- This adjunction is monoidal: Theorem 17 (p. 139)
- Separated automata have reduced state space: Example 36 (p. 147)

This is based on a paper under submission:

Moerman, J. & Rot, J. (2019). *Separation and Renaming in Nominal Sets*. (Under submission).

Besides these chapters in this thesis, I have published the following papers. These are not included in this thesis, but a short summary of those papers is presented below.

Complementing Model Learning with Mutation-Based Fuzzing. Our group at the Radboud University participated in the RERS challenge 2016. This is a challenge where reactive software is provided and researchers have to asses validity of certain properties (given as LTL specifications). We approached this with model learning: Instead of analysing the source code, we simply learned the external behaviour, and then used model checking on the learned model. This has worked remarkably well, as the models of the external behaviour are not too big. Our results were presented at the RERS workshop (ISOLA 2016). The report can be found on arXiv:
Smetsers, R., Moerman, J., Janssen, M., & Verwer, S. (2016). Complementing Model Learning with Mutation-Based Fuzzing. *CoRR*, *abs/1611.02429*. Retrieved from http://arxiv.org/abs/1611.02429.

π-Complete test suites for IOCO. In this paper, we investigate complete test suites for labelled transition systems (LTSs), instead of deterministic Mealy machines. This is a much harder problem than conformance testing of deterministic systems. The system may adversarially avoid certain states the tester wishes to test. We provide a test suite which is π-complete (provided the implementation is a suspension automaton). My main personal contribution here is the proof of completeness, which resembles the proof presented in Chapter 2 closely. The conference paper was presented at ICTSS:
van den Bos, P., Janssen, R., & Moerman, J. (2017). π-Complete Test Suites for IOCO. In *ICTSS 2017 Proceedings*. Springer. doi:10.1007/978-3-319-67549-7_6.
An extended version has appeared in:
van den Bos, P., Janssen, R., & Moerman, J. (2018). π-Complete Test Suites for IOCO. *Software Quality Journal*. Advanced online publication. doi:10.1007/s11219-018-9422-x.

Learning Product Automata. In this article, we consider Moore machines with multiple outputs. These machines can be decomposed by projecting on each output, resulting in smaller components that can be learned with fewer queries. We give experimental evidence that this is a useful technique which can reduce the number of queries substantially. This is all motivated by the idea that compositional methods are widely used throughout engineering and that we should use this in model learning. This work was presented at ICGI 2018:

Moerman, J. (2019). Learning Product Automata. In *International Conference on Grammatical Inference, ICGI, Proceedings*. Proceedings of Machine Learning Research. (To appear).

7 Conclusion and Outlook

With the current tools for model learning, it is possible to learn big state machines of black box systems. It involves using the clever algorithms for learning (such as the TTT algorithm by Isberner, 2015) and efficient testing methods (see Chapter 2). However, as the industrial case study from Chapter 3 shows, the bottleneck is often in conformance testing.

In order to improve on this bottleneck, one possible direction is to consider 'grey box testing.' The methods discussed in this thesis are all black box methods, this could be considered as 'too pessimistic.' Often, we do have (parts of the) source code and we do know relationships between different inputs. A question for future research is how this additional information can be integrated in a principled manner in the learning and testing of systems.

Black box testing still has theoretical challenges. Current generalisations to non-deterministic systems or language inclusion (such as black box testing for IOCO) often need exponentially big test suites. Whether this is necessary is unknown (to me): we only have upper bounds but no lower bounds. An interesting approach could be to see if there exists a notion of reduction between test suites. This is analogous to the reductions used in complexity theory to prove hardness of problems, or reductions used in PAC theory to prove learning problems to be inherently unpredictable.

Another path taken in this thesis is the research on nominal automata. This was motivated by the problem of learning automata over infinite alphabets. So far, the results on nominal automata are mostly theoretical in nature. Nevertheless, we show that the nominal algorithms can be implemented and that they can be run concretely on black box systems (Chapter 5). The advantage of using the foundations of nominal sets is that the algorithms are closely related to the original L* algorithm. Consequently, variations of L* can easily be implemented. For instance, we show that NL* algorithm for non-deterministic automata works in the nominal case too. (We have not attempted to implement more recent algorithms such as TTT.) The nominal learning algorithms can be implemented in just a few hundreds lines of code, much less than the approach taken by, e.g., Fiterău-Broștean (2018).

In this thesis, we tackle some efficiency issues when computing with nominal sets. In Chapter 6 we characterise orbits in order to give an efficient representation (for the total-order symmetry). Another result is the fact that some nominal automata can be 'compressed' to separated automata, which can be exponentially smaller (Chapter 7). However, the nominal tools still leave much to be desired in terms of efficiency.

Last, it would be interesting to marry the two paths taken in this thesis. I am not aware of n-complete test suites for register automata or nominal automata. The results on learning nominal automata in Chapter 5 show that this should be possible, as an observation table gives a test suite.[7] However, there is an interesting twist to this problem. The test methods from Chapter 2 can all account for extra states. For nominal automata, we should be able to cope with extra states *and* extra registers. It would be interesting to see how the test suite grows as these two dimensions increase.

[7] The rows of a table are access sequences, and the columns provide a characterisation set.

Part 1:
Testing Techniques

Chapter 2
FSM-based Test Methods

In this chapter, we will discuss some of the theory of test generation methods for black box conformance checking. Since the systems we consider are black box, we cannot simply determine equivalence with a specification. The only way to gain confidence is to perform experiments on the system. A key aspect of test generation methods is the size and completeness of the test suites. On one hand, we want to cover as much as the specification as possible, hopefully ensuring that we find mistakes in any faulty implementation. On the other hand: testing takes time, so we want to minimise the size of a test suite.

The test methods described here are well-known in the literature of FSM-based testing. They all share similar concepts, such as *access sequences* and *state identifiers*. In this chapter we will define these concepts, relate them with one another and show how to build test suites from these concepts. This theoretically discussion is new and enables us to compare the different methods uniformly. For instance, we can prove all these methods to be n-complete with a single proof.

The discussion also inspired a new algorithm: the *hybrid ADS methods*. This method is applied to an industrial case study in Chapter 3. It combines the strength of the ADS method (which is not always applicable) with the generality of the HSI method.

This chapter starts with the basics: Mealy machines, sequences and what it means to test a black box system. Then, starting from Section 1.3 we define several concepts, such as state identifiers, in order to distinguish one state from another. These concepts are then combined in Section 2 to derive test suites. In a similar vein, we define a novel test method in Section 3 and we discuss some of the implementation details of the `hybrid-ads` tool. We summarise the various test methods in Section 4. All methods are proven to be n-complete in Section 5. Finally, in Section 6, we discuss related work.

1 *Mealy machines and sequences*

We will focus on Mealy machines, as those capture many protocol specifications and reactive systems.

We fix finite alphabets I and O of inputs respectively outputs. We use the usual notation for operations on *sequences* (also called *words*): uv for the concatenation of two sequences $u, v \in I^*$ and $|u|$ for the length of u. For a sequence $w = uv$ we say that u and v are a prefix and suffix respectively.

Definition 1. A (deterministic and complete) *Mealy machine* M consists of a finite set of *states* S, an *initial state* $s_0 \in S$ and two functions:
– a *transition function* $\delta : S \times I \to S$, and
– an *output function* $\lambda : S \times I \to O$.
Both the transition function and output function are extended inductively to sequences as $\delta : S \times I^* \to S$ and $\lambda : S \times I^* \to O^*$:

$$\begin{aligned} \delta(s, \epsilon) &= s & \lambda(s, \epsilon) &= \epsilon \\ \delta(s, aw) &= \delta(\delta(s, a), w) & \lambda(s, aw) &= \lambda(s, a)\lambda(\delta(s, a), w) \end{aligned}$$

The *behaviour* of a state s is given by the output function $\lambda(s, -) : I^* \to O^*$. Two states s and t are *equivalent* if they have equal behaviours, written $s \sim t$, and two Mealy machines are equivalent if their initial states are equivalent.

Remark 2. We will use the following conventions and notation. We often write $s \in M$ instead of $s \in S$ and for a second Mealy machine M' its constituents are denoted S', s'_0, δ' and λ'. Moreover, if we have a state $s \in M$, we silently assume that s is not a member of any other Mealy machine M'. (In other words, the behaviour of s is determined by the state itself.) This eases the notation since we can write $s \sim t$ without needing to introduce a context.

An example Mealy machine is given in Figure 2.1.

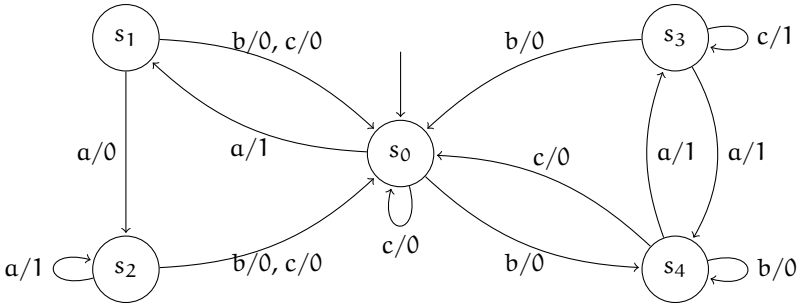


Figure 2.1 An example specification with input $I = \{a, b, c\}$ and output $O = \{0, 1\}$.

1.1 Testing

In conformance testing we have a specification modelled as a Mealy machine and an implementation (the system under test, or SUT) which we assume to behave as a Mealy machine. Tests, or experiments, are generated from the specification and applied to the implementation. We assume that we can reset the implementation before every test. If the output is different from the specified output, then we know

the implementation is flawed. The goals is to test as little as possible, while covering as much as possible.

A test suite is nothing more than a set of sequences. We do not have to encode outputs in the test suite, as those follow from the deterministic specification.

Definition 3. A *test suite* is a finite subset $T \subseteq I^*$.

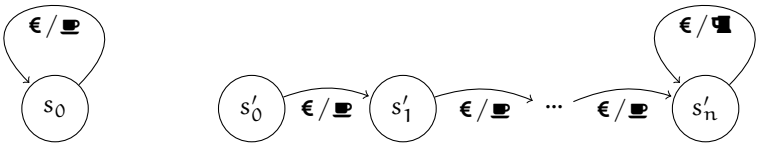
A test $t \in T$ is called *maximal* if it is not a proper prefix of another test $s \in T$. We denote the set of maximal tests of T by $max(T)$. The maximal tests are the only tests in T we actually have to apply to our SUT as we can record the intermediate outputs. In the examples of this chapter we will show $max(T)$ instead of T.

We define the size of a test suite as usual (Dorofeeva, et al., 2010 and Petrenko, et al., 2014). The size of a test suite is measured as the sum of the lengths of all its maximal tests plus one reset per test.

Definition 4. The *size* of a test suite T is defined to be $\|T\| = \sum_{t \in max(T)} (|t| + 1)$.

1.2 *Completeness of test suites*

Example 5. No test suite is complete. Consider the specification in Figure 2.2a. This machine will always outputs a cup of coffee – when given money. For any test suite we can make a faulty implementation which passes the test suite. A faulty implementation might look like Figure 2.2b, where the machine starts to output beers after n steps (signalling that it's the end of the day), where n is larger than the length of the longest sequence in the suite. This shows that no test-suite can be complete and it justifies the following definition.



(a) (b)

Figure 2.2 A basic example showing that finite test suites are incomplete. The implementation on the right will pass any test suite if we choose n big enough.

Definition 6. Let M be a Mealy machine and T be a test suite. We say that T is m-*complete* (*for* M) if for all inequivalent machines M' with at most m states there exists a $t \in T$ such that $\lambda(s_0, t) \neq \lambda'(s'_0, t)$.

We are often interested in the case of m-completeness, where $m = n + k$ for some $k \in \mathbb{N}$ and n is the number of states in the specification. Here k will stand for the number of *extra states* we can test.

Note the order of the quantifiers in the above definition. We ask for a single test suite which works for all implementations of bounded size. This is crucial for black box testing, as we do not know the implementation, so the test suite has to work for all of them.

1.3 Separating Sequences

Before we construct test suites, we discuss several types of useful sequences. All the following notions are standard in the literature, and the corresponding references will be given in Section 2, where we discuss the test generation methods using these notions. We fix a Mealy machine M for the remainder of this chapter.

Definition 7. We define the following kinds of sequences.
- Given two states s, t in M we say that w is a *separating sequence* if $\lambda(s, w) \neq \lambda(t, w)$.
- For a single state s in M, a sequence w is a *unique input output sequence* (*UIO*) if for every inequivalent state t in M we have $\lambda(s, w) \neq \lambda(t, w)$.
- Finally, a (*preset*) *distinguishing sequence* (*DS*) is a single sequence w which separates all states of M, i.e., for every pair of inequivalent states s, t in M we have $\lambda(s, w) \neq \lambda(t, w)$.

The above list is ordered from weaker to stronger notions, i.e., every distinguishing sequence is an UIO sequence for every state. Similarly, an UIO for a state s is a separating sequence for s and any inequivalent t. Separating sequences always exist for inequivalent states and finding them efficiently is the topic of Chapter 4. On the other hand, UIOs and DSs do not always exist for a machine.

A machine M is *minimal* if every distinct pair of states is inequivalent (i.e., $s \sim t \implies s = t$). We will not require M te be minimal, although this is often done in literature. Minimality is sometimes convenient, as one can write 'every other state t' instead of 'every inequivalent state t'.

Example 8. For the machine in Figure 2.1, we note that state s_0 and s_2 are separated by the sequence aa (but not by any shorter sequence). In fact, the sequence aa is an UIO for state s_0 since it is the only state outputting 10 on that input. However, state s_2 has no UIO: If the sequence were to start with b or c, state s_3 and s_4 respectively have equal transition, which makes it impossible to separate those states after the first symbol. If it starts with an a, states s_3 and s_4 are swapped and we make no progress in distinguishing these states from s_2. Since s_2 has no UIO, the machine as a whole does not admit a DS.

In this example, all other states actually have UIOs. For the states s_0, s_1, s_3 and s_4, we can pick the sequences aa, a, c and ac respectively. In order to separate s_2 from the other state, we have to pick multiple sequences. For instance, the set $\{aa, ac, c\}$ will separate s_2 from all other states.

1.4 Sets of separating sequences

As the example shows, we need sets of sequences and sometimes even sets of sets of sequences – called *families*.[8]

Definition 9. We define the following kinds of sets of sequences. We require that all sets are *prefix-closed*, however, we only show the maximal sequences in examples.[9]

- A set of sequences W is a called a *characterisation set* if it contains a separating sequence for each pair of inequivalent states in M.
- A *state identifier* for a state $s \in M$ is a set W_s such that for every inequivalent $t \in M$ a separating sequence for s and t exists in W_s.
- A set of state identifiers $\{W_s\}_s$ is *harmonised* if $W_s \cap W_t$ contains a separating sequence for inequivalent states s and t. This is also called a *separating family*.

A state identifier W_s will be used to test against a single state. In contrast to a characterisation set, it only include sequences which are relevant for s. The property of being harmonised might seem a bit strange. This property ensures that the same tests are used for different states. This extra consistency within a test suite is necessary for some test methods. We return to this notion in more detail in Example 22.

We may obtain a characterisation set by simply considering every pair of states and look for a difference. However, it turns out a harmonised set of state identifiers exists for every machine and this can be constructed very efficiently (Chapter 4). From a set of state identifiers we may obtain a characterisation set by taking the union of all those sets.

Example 10. As mentioned before, state s_2 from Figure 2.1 has a state identifier $\{aa, ac, b\}$. In fact, this set is a characterisation set for the whole machine. Since the other states have UIOs, we can pick singleton sets as state identifiers. For example, state s_0 has the UIO aa, so a state identifier for s_0 is $W_0 = \{aa\}$. Similarly, we can take $W_1 = \{a\}$ and $W_3 = \{c\}$. But note that such a family will *not* be harmonised since the sets $\{a\}$ and $\{c\}$ have no common separating sequence.

One more type of state identifier is of our interest: the *adaptive distinguishing sequence*. It it the strongest type of state identifier, and as a result not many machines have one. Like DSs, adaptive distinguishing sequences can identify a state using a single word. We give a slightly different (but equivalent) definition than the one of Lee and Yannakakis (1994).

Definition 11. A separating family $\mathcal{H}$ is an *adaptive distinguishing sequence* (ADS) if each set $\max(H_s)$ is a singleton.

[8] A family is often written as $\{X_s\}_{s \in M}$ or simply $\{X_s\}_s$, meaning that for each state $s \in M$ we have a set X_s.

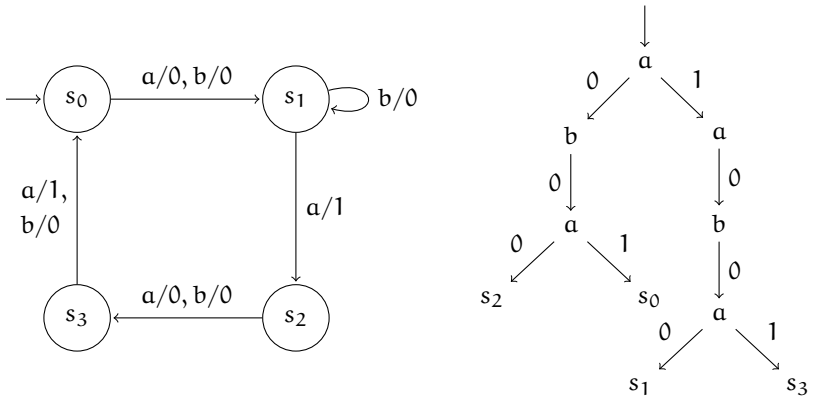
[9] Taking these sets to be prefix-closed makes many proofs easier.

It is called an adaptive sequence, since it has a tree structure which depends on the output of the machine. To see this tree structure, consider the first symbols of each of the sequences in the family. Since the family is harmonised and each set is essentially given by a single word, there is only one first symbol. Depending on the output after the first symbol, the sequence continues.

Example 12. In Figure 2.3 we see a machine with an ADS. The ADS is given as follows:

$$H_0 = \{aba\} \qquad H_1 = \{aaba\} \qquad H_2 = \{aba\} \qquad H_3 = \{aaba\}$$

Note that all sequences start with a. This already separates s_0, s_2 from s_1, s_3. To further separate the states, the sequences continues with either a b or another a. And so on.



(a) (b)

Figure 2.3 (a): A Mealy machine with an ADS and (b): the tree structure of this ADS.

Given an ADS, there exists an UIO for every state. The converse – if every state has an UIO, then the machine admits an ADS – does not hold. The machine in Figure 2.1 admits no ADS, since s_2 has no UIO.

1.5 *Partial equivalence*

Definition 13. We define the following notation.

- Let W be a set of sequences. Two states x, y are *W-equivalent*, written $x \sim_W y$, if $\lambda(x, w) = \lambda(y, w)$ for all $w \in W$.
- Let $\mathcal{W}$ be a family. Two states x, y are *$\mathcal{W}$-equivalent*, written $x \sim_{\mathcal{W}} y$, if $\lambda(x, w) = \lambda(y, w)$ for all $w \in W_x \cap W_y$.

The relation $\sim_W$ is an equivalence relation and $W \subseteq V$ implies that V separates more states than W, i.e., $x \sim_V y \implies x \sim_W y$. Clearly, if two states are equivalent (i.e., $s \sim t$), then for any set W we have $s \sim_W t$.

Lemma 14. The relations $\sim_W$ and $\sim_{\mathcal{W}}$ can be used to *define* characterisation sets and separating families. Concretely:
- W is a characterisation set if and only if for all s, t in M, $s \sim_W t$ implies $s \sim t$.
- $\mathcal{W}$ is a separating family if and only if for all s, t in M, $s \sim_{\mathcal{W}} t$ implies $s \sim t$.

Proof.
- W is a characterisation set by definition means $s \not\sim t \implies s \not\sim_W t$ as W contains a separating sequence (if it exists at all). This is equivalent to $s \sim_W t \implies s \sim t$.
- Let $\mathcal{W}$ be a separating family and $s \not\sim t$. Then there is a sequence $w \in W_s \cap W_t$ such that $\lambda(s, w) \neq \lambda(t, w)$, i.e., $s \not\sim_{\mathcal{W}} t$. We have shown $s \not\sim t \implies s \not\sim_{\mathcal{W}} t$, which is equivalent to $s \sim_{\mathcal{W}} t \implies s \sim t$. The converse is proven similarly. □

1.6 Access sequences

Besides sequences which separate states, we also need sequences which brings a machine to specified states.

Definition 15. An *access sequence for* s is a word w such that $\delta(s_0, w) = s$. A set P consisting of an access sequence for each state is called a *state cover*. If P is a state cover, then the set $\{pa \mid p \in P, a \in I\}$ is called a *transition cover*.

1.7 Constructions on sets of sequences

In order to define a test suite modularly, we introduce notation for combining sets of words. For sets of words X and Y, we define
- their concatenation $X \cdot Y = \{xy \mid x \in X, y \in Y\}$,
- iterated concatenation $X^0 = \{\epsilon\}$ and $X^{n+1} = X \cdot X^n$, and
- bounded concatenation $X^{\leqslant n} = \cup_{i \leqslant n} X^i$.

On families we define
- flattening: $\cup \mathcal{X} = \{x \mid x \in X_s, s \in S\}$,
- union: $\mathcal{X} \cup \mathcal{Y}$ is defined point-wise: $(\mathcal{X} \cup \mathcal{Y})_s = X_s \cup Y_s$,
- concatenation[10]: $X \odot \mathcal{Y} = \{xy \mid x \in X, y \in Y_{\delta(s_0, x)}\}$, and
- refinement: $\mathcal{X}; \mathcal{Y}$ defined by[11]

[10] We will often see the combination $P \cdot I \odot \mathcal{X}$, this should be read as $(P \cdot I) \odot \mathcal{X}$.

[11] We use the convention that $\cap$ binds stronger than $\cup$. In fact, all the operators here bind stronger than $\cup$.

$$(\mathcal{X};\mathcal{Y})_s = X_s \cup Y_s \cap \bigcup_{\substack{s \sim_{\mathcal{X}} t \\ s \not\sim_{\mathcal{Y}} t}} Y_t.$$

The latter construction is new and will be used to define a hybrid test generation method in Section 3. It refines a family $\mathcal{X}$, which need not be separating, by including sequences from a second family $\mathcal{Y}$. It only adds those sequences to states if $\mathcal{X}$ does not distinguish those states. This is also the reason behind the ;-notation: first the tests from $\mathcal{X}$ are used to distinguish states, *and then* for the remaining states $\mathcal{Y}$ is used.

Lemma 16. For all families $\mathcal{X}$ and $\mathcal{Y}$:
- $\mathcal{X};\mathcal{X} = \mathcal{X}$,
- $\mathcal{X};\mathcal{Y} = \mathcal{X}$, whenever $\mathcal{X}$ is a separating family, and
- $\mathcal{X};\mathcal{Y}$ is a separating family whenever $\mathcal{Y}$ is a separating family.

Proof. For the first item, note that there are no states t such that $s \sim_{\mathcal{X}} t$ and $s \not\sim_{\mathcal{X}} t$. Consequently, the union is empty, and the expression simplifies to

$$(\mathcal{X};\mathcal{X})_s = X_s \cup (X_s \cap \emptyset) = X_s.$$

If $\mathcal{X}$ is a separating family, then the only t for which $s \sim_{\mathcal{X}} t$ hold are t such that $s \sim t$ (Lemma 14). But $s \sim t$ is ruled out by $s \not\sim_{\mathcal{Y}} t$, and again so

$$(\mathcal{X};\mathcal{Y})_s = X_s \cup (Y_s \cap \emptyset) = X_s.$$

For the last item, suppose that $s \sim_{\mathcal{X};\mathcal{Y}} t$. Then s and t agree on every sequence in $(\mathcal{X};\mathcal{Y})_s \cap (\mathcal{X};\mathcal{Y})_t$. We distinguish two cases:
- Suppose $s \sim_{\mathcal{X}} t$, then $Y_s \cap Y_t \subseteq (\mathcal{X};\mathcal{Y})_s \cap (\mathcal{X};\mathcal{Y})_t$. And so s and t agree on $Y_s \cap Y_t$, meaning $s \sim_{\mathcal{Y}} t$. Since $\mathcal{Y}$ is a separating family, we have $s \sim t$.
- Suppose $s \not\sim_{\mathcal{X}} t$. This contradicts $s \sim_{\mathcal{X};\mathcal{Y}} t$, since $X_s \cap X_t \subseteq (\mathcal{X};\mathcal{Y})_s \cap (\mathcal{X};\mathcal{Y})_t$.

We conclude that $s \sim t$. This proves that $\mathcal{X};\mathcal{Y}$ is a separating family. □

2 *Test generation methods*

In this section, we review the classical conformance testing methods: the W, Wp, UIO, UIOv, HSI, ADS methods. At the end of this section, we construct the test suites for the running example. Our hybrid ADS method uses a similar construction.

There are many more test generation methods. Literature shows, however, that not all of them are complete. For example, the method by Bernhard (1994) is falsified by Petrenko (1997), and the UIO-method from Sabnani and Dahbura (1988) is shown to be incomplete by Chan, et al. (1989). For that reason, completeness of the correct methods is shown in Theorem 26. The proof is general enough to capture all the methods at once. We fix a state cover P throughout this section and take the transition cover $Q = P \cdot I$.

2.1 *W-method (Chow, 1978 and Vasilevskii, 1973)*

After the work of Moore (1956), it was unclear whether a test suite of polynomial size could exist. He presented a finite test suite which was complete, however it was exponential in size. Both Chow (1978) and Vasilevskii (1973) independently prove that test suites of polynomial size exist.[12] The W-method is a very structured test suite construction. It is called the W-method as the characterisation set is often called W.

Definition 17. Given a characterisation set W, we define the *W test suite* as

$$T_W = (P \cup Q) \cdot I^{\leqslant k} \cdot W.$$

This – and all following methods – tests the machine in two phases. For simplicity, we explain these phases when $k = 0$. The first phase consists of the tests $P \cdot W$ and tests whether all states of the specification are (roughly) present in the implementation. The second phase is $Q \cdot W$ and tests whether the successor states are correct. Together, these two phases put enough constraints on the implementation to know that the implementation and specification coincide (provided that the implementation has no more states than the specification).

2.2 *The Wp-method (Fujiwara, et al., 1991)*

Fujiwara, et al. (1991) realised that one needs fewer tests in the second phase of the W-method. Since we already know the right states are present after phase one, we only need to check if the state after a transition is consistent with the expected state. This justifies the use of state identifiers for each state.

Definition 18. Let $\mathcal{W}$ be a family of state identifiers. The *Wp test suite* is defined as

$$T_{Wp} = P \cdot I^{\leqslant k} \cdot \bigcup \mathcal{W} \cup Q \cdot I^{\leqslant k} \odot \mathcal{W}.$$

Note that $\cup \mathcal{W}$ is a characterisation set as defined for the W-method. It is needed for completeness to test states with the whole set $\cup \mathcal{W}$. Once states are tested as such, we can use the smaller sets W_s for testing transitions.

2.3 *The HSI-method (Luo, et al., 1995 and Petrenko, et al., 1993)*

The Wp-method in turn was refined by Luo, et al. (1995) and Petrenko, et al. (1993). They make use of harmonised state identifiers, allowing to take state identifiers in the initial phase of the test suite.

Definition 19. Let $\mathcal{H}$ be a separating family. We define the *HSI test suite* by

[12] More precisely: the size of T_W is polynomial in the size of the specification for each fixed k.

$$T_{\text{HSI}} = (P \cup Q) \cdot I^{\leqslant k} \odot \mathcal{H}.$$

Our hybrid ADS method is an instance of the HSI-method as we define it here. However, Luo, et al. (1995) and Petrenko, et al. (1993) describe the HSI-method together with a specific way of generating the separating families. Namely, the set obtained by a splitting tree with shortest witnesses. The hybrid ADS method does not refine the HSI-method defined in the more restricted sense.

2.4 *The ADS-method (Lee & Yannakakis, 1994)*

As discussed before, when a Mealy machine admits a adaptive distinguishing sequence, only a single test has to be performed for identifying a state. This is exploited in the ADS-method.

Definition 20. Let Z be an adaptive distinguishing sequence. The *ADS test suite* is defined as

$$T_{\text{ADS}} = (P \cup Q) \cdot I^{\leqslant k} \odot Z.$$

2.5 *The UIOv-method (Chan, et al., 1989)*

Some Mealy machines which do not admit an adaptive distinguishing sequence, may still admit state identifiers which are singletons. These are exactly UIO sequences and gives rise to the UIOv-method. In a way this is a generalisation of the ADS-method, since the requirement that state identifiers are harmonised is dropped.

Definition 21. Let $U = \{\text{a single UIO for } s\}_{s \in S}$ be a family of UIO sequences, the *UIOv test suite* is defined as

$$T_{\text{UIOv}} = P \cdot I^{\leqslant k} \cdot \bigcup U \cup Q \cdot I^{\leqslant k} \odot U.$$

One might think that using a single UIO sequence instead of the set $\cup U$ to verify the state is enough. In fact, this idea was used for the *UIO-method* which defines the test suite $(P \cup Q) \cdot I^{\leqslant k} \odot U$. The following is a counterexample, due to Chan, et al. (1989), to such conjecture.

Example 22. The Mealy machines in Figure 2.4 shows that UIO-method does not define a 3-complete test suite. Take for example the UIOs $u_0 = aa, u_1 = a, u_2 = ba$ for the states s_0, s_1, s_2 respectively. The test suite then becomes $\{aaaa, abba, baaa, bba\}$ and the faulty implementation passes this suite. This happens because the sequence u_2 is not an UIO in the implementation, and the state s_2' simulates both UIOs u_1 and u_2. Hence we also want to check that a state does not behave as one of the other states, and therefore we use $\cup U$. With the same UIOs as above, the resulting UIOv test suite for the specification in Figure 2.4 is $\{aaaa, aba, abba, baaa, bba\}$ of size 23. (Recall that we also count resets when measuring the size.)

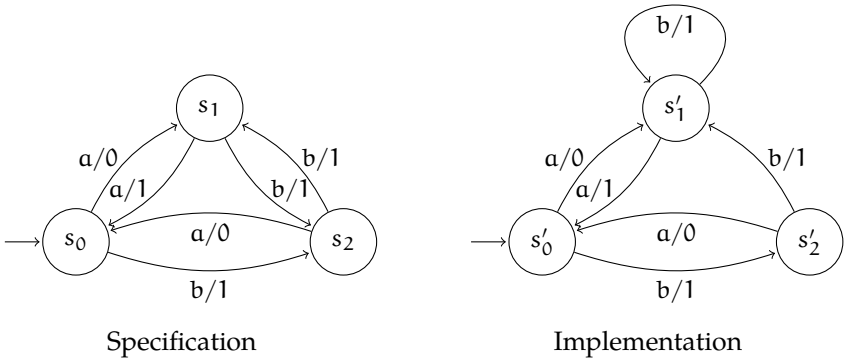


Figure 2.4 An example where the UIO-method is not complete.

2.6 *All test suites for Figure 2.1*

Let us compute all the previous test suites on the specification in Figure 2.1. We will be testing without extra states, i.e., we construct 5-complete test suites. We start by defining the state and transition cover. For this, we take all shortest sequences from the initial state to the other states. This state cover is depicted in Figure 2.5. The transition cover is simply constructed by extending each access sequence with another symbol.

$$P = \{\epsilon, a, aa, b, ba\}$$

$$Q = P \cdot I = \{a, b, c, aa, ab, ac, aaa, aab, aac, ba, bb, bc, baa, bab, bac\}$$

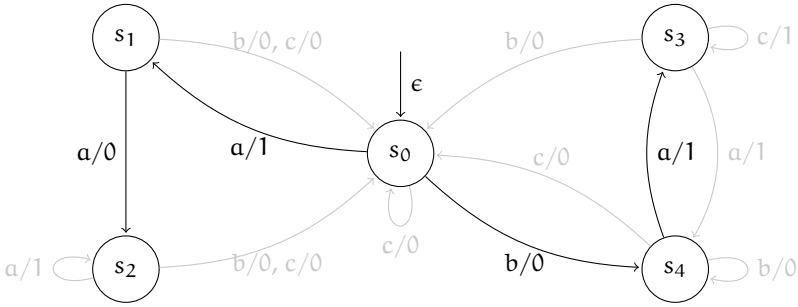


Figure 2.5 A state cover for the specification from Figure 2.1.

As shown earlier, the set $W = \{aa, ac, c\}$ is a characterisation set. The W-method, which simply combines $P \cup Q$ with W, gives the following test suite of size 169:

$$\begin{aligned} T_W = \{ \quad & aaaaa, aaaac, aaac, aabaa, aabac, aabc, aacaa, \\ & aacac, aacc, abaa, abac, abc, acaa, acac, acc, baaaa, \\ & baaac, baac, babaa, babac, babc, bacaa, bacac, bacc, \\ & bbaa, bbac, bbc, bcaa, bcac, bcc, caa, cac, cc \quad \} \end{aligned}$$

With the Wp-method we get to choose a different state identifier per state. Since many states have an UIO, we can use them as state identifiers. This defines the following family $\mathcal{W}$:

$$W_0 = \{aa\} \qquad W_1 = \{a\} \qquad W_2 = \{aa, ac, c\} \qquad W_3 = \{c\} \qquad W_4 = \{ac\}$$

For the first part of the Wp test suite we need $\cup\mathcal{W} = \{aa, ac, c\}$. For the second part, we only combine the sequences in the transition cover with the corresponding suffixes. All in all we get a test suite of size 75:

$$\begin{aligned} T_{Wp} = \{ \quad & aaaaa, aaaac, aaac, aabaa, aacaa, abaa, \\ & acaa, baaac, baac, babaa, bacc, bbac, bcaa, caa \quad \} \end{aligned}$$

For the HSI-method we need a separating family $\mathcal{H}$. We pick the following sets:

$$H_0 = \{aa, c\} \qquad H_1 = \{a\} \qquad H_2 = \{aa, ac, c\} \qquad H_3 = \{a, c\} \qquad H_4 = \{aa, ac, c\}$$

(We repeat that these sets are prefix-closed, but we only show the maximal sequences.) Note that these sets are harmonised, unlike the family $\mathcal{W}$. For example, the separating sequence a is contained in both H_1 and H_3. This ensures that we do not have to consider $\cup\mathcal{H}$ in the first part of the test suite. When combining this with the corresponding prefixes, we obtain the HSI test suite of size 125:

$$\begin{aligned} T_{HSI} = \{ \quad & aaaaa, aaaac, aaac, aabaa, aabc, aacaa, aacc, \\ & abaa, abc, acaa, acc, baaaa, baaac, baac, babaa, \\ & babc, baca, bacc, bbaa, bbac, bbc, bcaa, bcc, caa, cc \quad \} \end{aligned}$$

On this particular example the Wp-method outperforms the HSI-method. The reason is that many states have UIOs and we picked those to be the state identifiers. In general, however, UIOs may not exist (and finding them is hard).

The UIO-method and ADS-method are not applicable in this example because state s_2 does not have an UIO.

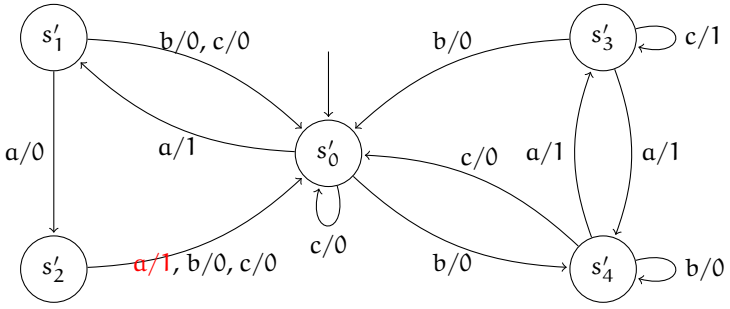


Figure 2.6 A faulty implementation for the specification in Figure 2.1.

We can run these test suites on the faulty implementation shown in Figure 2.6. Here, the a-transition from state s'_2 transitions to the wrong target state. It is not an obvious mistake, since the faulty target s'_0 has very similar transitions as s_2. Yet, all the test suites detect this error. When choosing the prefix aaa (included in the transition cover), and suffix aa (included in the characterisation set and state identifiers for s_2), we see that the specification outputs 10111 and the implementation outputs 10110. The sequence $aaaaa$ is the only sequence (in any of the test suites here) which detects this fault.

Alternatively, the a-transition from s'_2 would transition to s'_4, we need the suffix ac as aa will not detect the fault. Since the sequences ac is included in the state identifier for s_2, this fault would also be detected. This shows that it is sometimes necessary to include multiple sequences in the state identifier.

Another approach to testing would be to enumerate all sequences up to a certain length. In this example, we need sequences of at least length 5. Consequently, the test suite contains 243 sequences and this boils down to a size of 1458. Such a brute-force approach is not scalable.

3 *Hybrid ADS method*

In this section, we describe a new test generation method for Mealy machines. Its completeness will be proven in Theorem 26, together with completeness for all methods defined in the previous section. From a high level perspective, the method uses the algorithm by Lee and Yannakakis (1994) to obtain an ADS. If no ADS exists, their algorithm still provides some sequences which separates some inequivalent states. Our extension is to refine the set of sequences by using pairwise separating sequences. Hence, this method is a hybrid between the ADS-method and HSI-method.

The reason we do this is the fact that the ADS-method generally constructs small test suites as experiments by Dorofeeva, et al. (2010) suggest. The test suites are small since an ADS can identify a state with a single word, instead of a set of words which is generally needed. Even if the ADS does not exist, using the partial result of Lee and Yannakakis' algorithm can reduce the size of test suites.

We will now see the construction of this hybrid method. Instead of manipulating separating families directly, we use a *splitting tree*. This is a data structure which is used to construct separating families or adaptive distinguishing sequences.

Definition 23. A *splitting tree* (*for* M) is a rooted tree where each node u has
- a non-empty set of states $l(u) \subseteq M$, and
- if u is not a leaf, a sequence $\sigma(u) \in I^*$.

We require that if a node u has children $C(u)$ then
- the sets of states of the children of u partition $l(u)$, i.e., the set $P(u) = \{l(v) \mid v \in C(u)\}$ is a *non-trivial* partition of $l(u)$, and

– the sequence $\sigma(u)$ witnesses the partition $P(u)$, meaning that for all $p, q \in P(u)$ we have $p = q$ iff $\lambda(s, \sigma(u)) = \lambda(t, \sigma(u))$ for all $s \in p, t \in q$.

A splitting tree is called *complete* if all inequivalent states belong to different leaves.

Efficient construction of a splitting tree is described in more detail in Chapter 4. Briefly, the splitting tree records the execution of a partition refinement algorithm (such as Moore's or Hopcroft's algorithm). Each non-leaf node encodes a *split* together with a witness, which is a separating sequence for its children. From such a tree we can construct a state identifier for a state by locating the leaf containing that state and collecting all the sequences you read when traversing to the root.

For adaptive distinguishing sequences an additional requirement is put on the splitting tree: for each non-leaf node u, the sequence $\sigma(u)$ defines an injective map $x \mapsto (\delta(x, \sigma(u)), \lambda(x, \sigma(u)))$ on the set $l(u)$. Lee and Yannakakis (1994) call such splits *valid*. Figure 2.7 shows both valid and invalid splits. Validity precisely ensures that after performing a split, the states are still distinguishable. Hence, sequences of such splits can be concatenated.

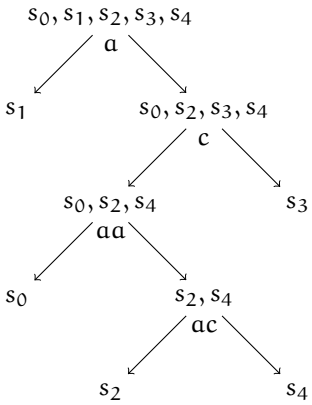


Figure 2.7 A complete splitting tree with shortest witnesses for the specification of Figure 2.1. Only the splits a, aa, and ac are valid.

The following lemma is a result of Lee and Yannakakis (1994).

Lemma 24. A complete splitting tree with only valid splits exists if and only if there exists an adaptive distinguishing sequence.

Our method uses the exact same algorithm as the one by Lee and Yannakakis. However, we also apply it in the case when the splitting tree with valid splits is not complete (and hence no adaptive distinguishing sequence exists). Their algorithm still produces a family of sets, but is not necessarily a separating family.

In order to recover separability, we refine that family. Let $\mathcal{Z}'$ be the result of Lee and Yannakakis' algorithm (to distinguish from their notation, we add a prime) and let $\mathcal{H}$ be a separating family extracted from an ordinary splitting tree. The hybrid ADS family is defined as $\mathcal{Z}';\mathcal{H}$, and can be computed as sketched in Algorithm 2.1 (the algorithm works on splitting trees instead of separating families). By Lemma 16 we note the following: in the best case this family is an adaptive distinguishing sequence; in the worst case it is equal to $\mathcal{H}$; and in general it is a combination of the two families. In all cases, the result is a separating family because $\mathcal{H}$ is.

Require: A Mealy machine M
Ensure: A separating family Z
1 $T_1 \leftarrow$ splitting tree for Moore's minimisation algorithm
2 $T_2 \leftarrow$ splitting tree with valid splits (see Lee & Yannakakis, 1994)
3 $\mathcal{Z}' \leftarrow$ (incomplete) family constructed from T_2
4 **for all** inequivalent states s, t in the same leaf of T_2 **do**
5 $\quad u \leftarrow \mathrm{lca}(T_1, s, t)$
6 $\quad Z_s \leftarrow Z'_s \cup \{\sigma(u)\}$
7 $\quad Z_t \leftarrow Z'_t \cup \{\sigma(u)\}$
8 **end for**
9 **return** Z

Algorithm 2.1 Obtaining the hybrid separating family $\mathcal{Z}';\mathcal{H}$

With the hybrid family we can define the test suite as follows. Its m-completeness is proven in Section 5.

Definition 25. Let P be a state cover, $\mathcal{Z}'$ be a family of sets constructed with the Lee and Yannakakis algorithm, and $\mathcal{H}$ be a separating family. The *hybrid ADS* test suite is

$$T_{\text{h-ADS}} = (P \cup Q) \cdot I^{\leqslant k} \odot (\mathcal{Z}';\mathcal{H}).$$

3.1 Example

In the figure we see the (unique) result of Lee and Yannakakis' algorithm. We note that the states s_2, s_3, s_4 are not split, so we need to refine the family for those states.

We take the separating family $\mathcal{H}$ from before. From the incomplete ADS in Figure 2.8b above we obtain the family $\mathcal{Z}'$. These families and the refinement $\mathcal{Z}';\mathcal{H}$ are given below.

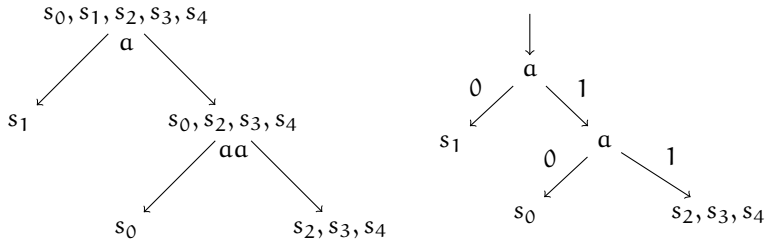


Figure 2.8 (a): Largest splitting tree with only valid splits for Figure 2.1. (b): Its incomplete adaptive distinguishing tree.

$$
\begin{array}{lll}
H_0 = \{aa, c\} & Z'_0 = \{aa\} & (Z';H)_0 = \{aa\} \\
H_1 = \{a\} & Z'_1 = \{a\} & (Z';H)_1 = \{a\} \\
H_2 = \{aa, ac, c\} & Z'_2 = \{aa\} & (Z';H)_2 = \{aa, ac, c\} \\
H_3 = \{a, c\} & Z'_3 = \{aa\} & (Z';H)_3 = \{aa, c\} \\
H_4 = \{aa, ac, c\} & Z'_4 = \{aa\} & (Z';H)_4 = \{aa, ac, c\}
\end{array}
$$

With the separating family $\mathcal{Z}';\mathcal{H}$ we obtain the following test suite of size 96:

$$
\begin{aligned}
T_{\text{h-ADS}} = \{ \quad & aaaaa, aaaac, aaac, aabaa, aacaa, abaa, acaa, \\
& baaaa, baaac, baac, babaa, bacaa, bacc, bbaa, bbac, \\
& bbc, bcaa, caa \quad \}
\end{aligned}
$$

We note that this is indeed smaller than the HSI test suite. In particular, we have a smaller state identifier for s_0: $\{aa\}$ instead of $\{aa, c\}$. As a consequence, there are less combinations of prefixes and suffixes. We also observe that one of the state identifiers grew in length: $\{aa, c\}$ instead of $\{a, c\}$ for state s_3.

3.2 *Implementation*

All the algorithms concerning the hybrid ADS-method have been implemented and can be found at https://github.com/Jaxan/hybrid-ads. We note that Algorithm 2.1 is implemented a bit more efficiently, as we can walk the splitting trees in a particular order. For constructing the splitting trees in the first place, we use Moore's minimisation algorithm and the algorithms by Lee and Yannakakis (1994). We keep all relevant sets prefix-closed by maintaining a *trie data structure*. A trie data structure also allows us to immediately obtain the set of maximal tests only.

3.3 *Randomisation*

Many constructions of the test suite generation can be randomised. There may exist many shortest access sequences to a state and we can randomly pick any. Also in the

construction of state identifiers many steps in the algorithm are non-deterministic: the algorithm may ask to find any input symbol which separates a set of states. The tool randomises many such choices. We have noticed that this can have a huge influence on the size of the test suite. However, a decent statistical investigation still lacks at the moment.

In many of the applications such as learning, no bound on the number of states of the SUT is known. In such cases it is possible to randomly select test cases from an infinite test suite. Unfortunately, we lose the theoretical guarantees of completeness with random generation. Still, as we will see in Chapter 3, this can work really well. We can randomly test cases as follows. In the above definition for the hybrid ADS test suite we replace $I^{\leqslant k}$ by I^* to obtain an infinite test suite. Then we sample tests as follows:

1. sample an element p from P uniformly,
2. sample a word w from I^* with a geometric distribution, and
3. sample uniformly from $(\mathcal{Z}'; \mathcal{H})_s$ for the state $s = \delta(s_0, pw)$.

4 Overview

We give an overview of the aforementioned test methods. We classify them in two directions,

– whether they use harmonised state identifiers or not and
– whether it used singleton state identifiers or not.

Theorem 26. Assume M to be minimal, reachable, and of size n. The following test suites are all $n + k$-complete:

	Arbitrary	Harmonised
Many / pairwise	Wp $P \cdot I^{\leqslant k} \cdot \cup \mathcal{W} \cup Q \cdot I^{\leqslant k} \odot \mathcal{W}$	HSI $(P \cup Q) \cdot I^{\leqslant k} \odot \mathcal{H}$
Hybrid		Hybrid ADS $(P \cup Q) \cdot I^{\leqslant k} \odot (\mathcal{Z}'; \mathcal{H})$
Single / global	UIOv $P \cdot I^{\leqslant k} \cdot \cup \mathcal{U} \cup Q \cdot I^{\leqslant k} \odot \mathcal{U}$	ADS $(P \cup Q) \cdot I^{\leqslant k} \odot \mathcal{Z}$

Proof. See Corollary 33 and 35. □

Each of the methods in the right column can be written simpler as $P \cdot I^{\leqslant k+1} \odot \mathcal{H}$, since $Q = P \cdot I$. This makes them very easy to implement.

It should be noted that the ADS-method is a specific instance of the HSI-method and similarly the UIOv-method is an instance of the Wp-method. What is generally

meant by the Wp-method and HSI-method is the above formula together with a particular way to obtain the (harmonised) state identifiers.

We are often interested in the size of the test suite. In the worst case, all methods generate a test suite with a size in $\mathcal{O}(pn^3)$ and this bound is tight (Vasilevskii, 1973). Nevertheless we expect intuitively that the right column performs better, as we are using a more structured set (given a separating family for the HSI-method, we can always forget about the common prefixes and apply the Wp-method, which will never be smaller if constructed in this way). Also we expect the bottom row to perform better as there is a single test for each state. Small experimental results confirm this intuition (Dorofeeva, et al., 2010).

On the example in Figure 2.1, we computed all applicable test suites in Sections 2.6 and 3.1. The UIO and ADS methods are not applicable. For the W, Wp, HSI and hybrid ADS methods we obtained test suites of size 169, 75, 125 and 96 respectively.

5 *Proof of completeness*

In this section, we will prove n-completeness of the discussed test methods. Before we dive into the proof, we give some background on the proof-principle of bisimulation. The original proofs of completeness often involve an inductive argument (on the length of words) inlined with arguments about characterisation sets. This can be hard to follow and so we prefer a proof based on bisimulations, which defers the inductive argument to a general statement about bisimulation. Many notions of bisimulation exist in the theory of labelled transition systems, but for Mealy machines there is just one simple definition. We give the definition and the main proof principle, all of which can be found in a paper by Rutten (1998).

Definition 27. Let M be a Mealy machine. A relation $R \subseteq S \times S$ is called a *bisimulation* if for every $(s, t) \in R$ we have
- equal outputs: $\lambda(s, a) = \lambda(t, a)$ for all $a \in I$, and
- related successor states: $(\delta(s, a), \delta(t, a)) \in R$ for all $a \in I$.

Lemma 28. If two states s, t are related by a bisimulation, then $s \sim t$.[13]

We use a slight generalisation of the bisimulation proof technique, called *bisimulation up-to*. This allows one to give a smaller set R which extends to a bisimulation. A good introduction of these up-to techniques is given by Bonchi and Pous (2015) or the thesis of Rot (2015). In our case we use bisimulation *up-to* $\sim$*-union*. The following lemma can be found in the given references.

[13] The converse – which we do not need here – also holds, as $\sim$ is a bisimulation.

Definition 29. Let M be a Mealy machine. A relation $R \subseteq S \times S$ is called a *bisimulation up-to $\sim$-union* if for every $(s,t) \in R$ we have
- equal outputs: $\lambda(s,a) = \lambda(t,a)$ for all $a \in I$, and
- related successor states: $(\delta(s,a), \delta(t,a)) \in R$ *or* $\delta(s,a) \sim \delta(t,a)$ for all $a \in I$.

Lemma 30. Any bisimulation up-to $\sim$-union is contained in a bisimulation.

We fix a specification M which has a minimal representative with n states and an implementation M' with at most $n+k$ states. We assume that all states are reachable from the initial state in both machines (i.e., both are *connected*).

The next proposition gives sufficient conditions for a test suite of a certain shape to be complete. We then prove that these conditions hold for the test suites in this chapter.

Proposition 31. Let W and W' be two families of words and P a state cover for M. Let $T = P \cdot I^{\leqslant k} \odot W \cup P \cdot I^{\leqslant k+1} \odot W'$ be a test suite. If
1. for all $x, y \in M : x \sim_{W_x \cap W_y} y$ implies $x \sim y$,
2. for all $x, y \in M$ and $z \in M'$: $x \sim_{W_x} z$ and $z \sim_{W'_y} y$ implies $x \sim y$, and
3. the machines M and M' agree on T,

then M and M' are equivalent.

Proof. First, we prove that $P \cdot I^{\leqslant k}$ reaches all states in M'. For $p, q \in P$ and $x = \delta(s_0, p), y = \delta(s_0, q)$ such that $x \not\sim_{W_x \cap W_y} y$, we also have $\delta'(s'_0, p) \not\sim_{W_x \cap W_y} \delta'(s'_0, q)$ in the implementation M'. By (1) this means that there are at least n different behaviours in M', hence at least n states.

Now n states are reached by the previous argument (using the set P). By assumption M' has at most k extra states. If those extra states are reachable, they are reachable from an already visited state in at most k steps. So we can reach all states of M' by using $I^{\leqslant k}$ after P.

Second, we show that the reached states are bisimilar. Define the relation $R = \{(\delta(s_0, p), \delta'(s'_0, p)) \mid p \in P \cdot I^{\leqslant k}\}$. Note that for each $(s, i) \in R$ we have $s \sim_{W_s} i$. For each state $i \in M'$ there is a state $s \in M$ such that $(s, i) \in R$, since we reach all states in both machines by $P \cdot I^{\leqslant k}$. We will prove that this relation is in fact a bisimulation up-to $\sim$-union.

For output, we note that $(s, i) \in R$ implies $\lambda(s, a) = \lambda'(i, a)$ for all a, since the machines agree on $P \cdot I^{\leqslant k+1}$. For the successors, let $(s, i) \in R$ and $a \in I$ and consider the successors $s_2 = \delta(s, a)$ and $i_2 = \delta'(i, a)$. We know that there is some $t \in M$ with $(t, i_2) \in R$. We also know that we tested i_2 with the set W_t. So we have:

$$s_2 \sim_{W'_{s_2}} i_2 \sim_{W_t} t.$$

By the second assumption, we conclude that $s_2 \sim t$. So $s_2 \sim t$ and $(t, i) \in R$, which means that R is a bisimulation up-to $\sim$-union. Moreover, R contains the pair (s_0, s'_0). By using Lemmas 30 and 28, we conclude that the initial s_0 and s'_0 are equivalent. □

Before we show that the conditions hold for the test methods, we reflect on the above proof first. This proof is very similar to the completeness proof by Chow (1978).[14] In the first part we argue that all states are visited by using some sort of counting and reachability argument. Then in the second part we show the actual equivalence. To the best of the authors knowledge, this is first m-completeness proof which explicitly uses the concept of a bisimulation. Using a bisimulation allows us to slightly generalise and use bisimulation up-to $\sim$-union, dropping the the often-assumed requirement that M is minimal.

Lemma 32. Let $\mathcal{W}'$ be a family of state identifiers for M. Define the family $\mathcal{W}$ by $W_s = \cup \mathcal{W}'$. Then the conditions (1) and (2) in Proposition 31 are satisfied.

Proof. The first condition we note that $W_x \cap W_y = W_x = W_y$, and so $x \sim_{W_x \cap W_y} y$ implies $x \sim_{W_x} y$, now by definition of state identifier we get $x \sim y$.

For the second condition, let $x \sim_{\cup \mathcal{W}'} z \sim_{W'_y} y$. Then we note that $W'_y \subseteq \cup W'$ and so we get $x \sim_{W'_y} z \sim_{W'_y} y$. By transitivity we get $x \sim_{W'_y} y$ and so by definition of state identifier we get $x \sim y$. □

Corollary 33. The W, Wp, and UIOv test suites are $n + k$-complete.

Lemma 34. Let $\mathcal{H}$ be a separating family and take $\mathcal{W} = \mathcal{W}' = \mathcal{H}$. Then the conditions (1) and (2) in Proposition 31 are satisfied.

Proof. Let $x \sim_{H_x \cap H_y} y$, then by definition of separating family $x \sim y$. For the second condition, let $x \sim_{H_x} z \sim_{H_y} y$. Then we get $x \sim_{H_x \cap H_y} z \sim_{H_x \cap H_y} y$ and so by transitivity $x \sim_{H_x \cap H_y} y$, hence again $x \sim y$. □

Corollary 35. The HSI, ADS and hybrid ADS test suites are $n + k$-complete.

6 Related Work and Discussion

In this chapter, we have mostly considered classical test methods which are all based on prefixes and state identifiers. There are more recent methods which almost fit in the same framework. We mention the P (Simão & Petrenko, 2010), H (Dorofeeva, et al., 2005), and SPY (Simão, et al., 2009) methods. The P method construct a test suite by carefully considering sufficient conditions for a p-complete test suite (here $p \leqslant n$, where n is the number of states). It does not generalise to extra states, but it seems to construct very small test suites. The H method is a refinement of the HSI-method where state identifiers for a testing transitions are reconsidered. (Note that Proposition 31 allows for a different family when testing transitions.) Last, the SPY

[14] In fact, it is also similar to Lemma 4 by Angluin (1987) which proves termination in the L* learning algorithm. This correspondence was noted by Berg, et al. (2005).

method builds upon the HSI-method and changes the prefixes in order to minimise the size of a test suite, exploiting overlap in test sequences. We believe that this technique is independent of the HSI-method and can in fact be applied to all methods presented in this chapter. As such, the SPY method should be considered as an optimisation technique, orthogonal to the work in this chapter.

Recently, Hierons and Türker (2015) devise a novel test method which is based on incomplete distinguishing sequences and is similar to the hybrid ADS method. They use sequences which can be considered to be adaptive distinguishing sequences on a subset of the state space. With several of those one can cover the whole state space, obtaining a m-complete test suite. This is a bit dual to our approach, as our "incomplete" adaptive distinguishing sequences define a course partition of the complete state space. Our method becomes complete by refining the tests with pairwise separating sequences.

Some work is put into minimising the adaptive distinguishing sequences themselves. Türker and Yenigün (2014) describe greedy algorithms which construct small adaptive distinguishing sequences. Moreover, they show that finding the minimal adaptive distinguishing sequence is NP-complete in general, even approximation is NP-complete. We expect that similar heuristics also exist for the other test methods and that they will improve the performance. Note that minimal separating sequences do not guarantee a minimal test suite. In fact, we see that the hybrid ADS method outperforms the HSI-method on the example in Figure 2.1 since it prefers longer, but fewer, sequences.

Some of the assumptions made at the start of this chapter have also been challenged. For non-deterministic Mealy machine, we mention the work of Petrenko and Yevtushenko (2014). We also mention the work of van den Bos, et al. (2017) and Simão and Petrenko (2014) for input/output transition systems with the *ioco* relation. In both cases, the test suites are still defined in the same way as in this chapter: prefixes followed by state identifiers. However, for non-deterministic systems, guiding an implementation into a state is harder as the implementation may choose its own path. For that reason, sequences are often replaced by automata, so that the testing can be adaptive. This adaptive testing is game-theoretic and the automaton provides a strategy. This game theoretic point of view is further investigated by van den Bos and Stoelinga (2018). The test suites generally are of exponential size, depending on how non-deterministic the systems are.

The assumption that the implementation is resettable is also challenged early on. If the machine has no reliable reset (or the reset is too expensive), one tests the system with a single *checking sequence*. Lee and Yannakakis (1994) give a randomised algorithm for constructing such a checking sequence using adaptive distinguishing sequences. There is a similarity with the randomised algorithm by Rivest and Schapire (1993) for learning non-resettable automata. Recently, Groz, et al. (2018) give a deterministic learning algorithm for non-resettable machines based on adaptive distinguishing sequences.

Many of the methods described here are benchmarked on small or random Mealy machines by Dorofeeva, et al. (2010) and Endo and Simão (2013). The benchmarks are of limited scope, the machine from Chapter 3, for instance, is neither small nor random. For this reason, we started to collect more realistic benchmarks at http://automata.cs.ru.nl/.

Chapter 3
Applying Automata Learning to Embedded Control Software

Wouter Smeek
Océ Technologies B.V.

Joshua Moerman
Radboud University

Frits Vaandrager
Radboud University

David N. Jansen
Radboud University

Abstract

Using an adaptation of state-of-the-art algorithms for black-box automata learning, as implemented in the LearnLib tool, we succeeded to learn a model of the Engine Status Manager (ESM), a software component that is used in printers and copiers of Océ. The main challenge that we encountered was that LearnLib, although effective in constructing hypothesis models, was unable to find counterexamples for some hypotheses. In fact, none of the existing FSM-based conformance testing methods that we tried worked for this case study. We therefore implemented an extension of the algorithm of Lee & Yannakakis for computing an adaptive distinguishing sequence. Even when an adaptive distinguishing sequence does not exist, Lee & Yannakakis' algorithm produces an adaptive sequence that "almost" identifies states. In combination with a standard algorithm for computing separating sequences for pairs of states, we managed to verify states with on average 3 test queries. Altogether, we needed around 60 million queries to learn a model of the ESM with 77 inputs and 3.410 states. We also constructed a model directly from the ESM software and established equivalence with the learned model. To the best of our knowledge, this is the first paper in which active automata learning has been applied to industrial control software.

This chapter is based on the following publication:
Smeenk, W., Moerman, J., Vaandrager, F. W., & Jansen, D. N. (2015). Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM, Proceedings*. Springer. doi:10.1007/978-3-319-25423-4_5

Once they have high-level models of the behaviour of software components, software engineers can construct better software in less time. A key problem in practice, however, is the construction of models for existing software components, for which no or only limited documentation is available.

The construction of models from observations of component behaviour can be performed using regular inference – also known as automata learning (see Angluin, 1987; de la Higuera, 2010; Steffen, et al., 2011). The most efficient such techniques use the set-up of *active learning*, illustrated in Figure 3.1, in which a "learner" has the task to learn a model of a system by actively asking questions to a "teacher".

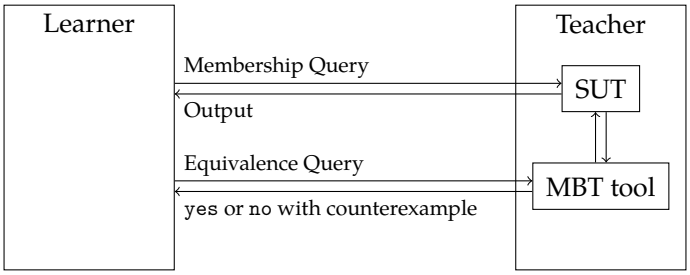


Figure 3.1 Active learning of reactive systems.

The core of the teacher is a *System Under Test* (SUT), a reactive system to which one can apply inputs and whose outputs one may observe. The learner interacts with the SUT to infer a model by sending inputs and observing the resulting outputs ("membership queries"). In order to find out whether an inferred model is correct, the learner may pose an "equivalence query". The teacher uses a model-based testing (MBT) tool to try and answer such queries: Given a hypothesised model, an MBT tool generates a long test sequence using some conformance testing method. If the SUT passes this test, then the teacher informs the learner that the model is deemed correct. If the outputs of the SUT and the model differ, this constitutes a counterexample, which is returned to the learner. Based on such a counterexample, the learner may then construct an improved hypothesis. Hence, the task of the learner is to collect data by interacting with the teacher and to formulate hypotheses, and the task of the MBT tool is to establish the validity of these hypotheses. It is important to note that it may occur that an SUT passes the test for an hypothesis, even though this hypothesis is not valid.

Triggered by various theoretical and practical results, see for instance the work by Aarts (2014); Berg, et al. (2005); Cassel, et al. (2015); Howar, et al. (2012); Leucker (2006); Merten, et al. (2012); Raffelt, et al. (2009), there is a fast-growing interest in automata learning technology. In recent years, automata learning has been applied successfully, e.g., to regression testing of telecommunication systems (Hungar, et al., 2003), checking conformance of communication protocols to a reference implementation (Aarts, et al., 2014), finding bugs in Windows and Linux implementations of TCP

(Fiterău-Broștean, et al., 2014), analysis of botnet command and control protocols (Cho, et al., 2010), and integration testing (Groz, et al., 2008 and Li, et al., 2006).

In this chapter, we explore whether LearnLib by Raffelt, et al. (2009), a state-of-the-art automata learning tool, is able to learn a model of the Engine Status Manager (ESM), a piece of control software that is used in many printers and copiers of Océ. Software components like the ESM can be found in many embedded systems in one form or another. Being able to retrieve models of such components automatically is potentially very useful. For instance, if the software is fixed or enriched with new functionality, one may use a learned model for regression testing. Also, if the source code of software is hard to read and poorly documented, one may use a model of the software for model-based testing of a new implementation, or even for generating an implementation on a new platform automatically. Using a model checker one may also study the interaction of the software with other components for which models are available.

The ESM software is actually well documented, and an extensive test suite exists. The ESM, which has been implemented using Rational Rose Real-Time (RRRT), is stable and has been in use for 10 years. Due to these characteristics, the ESM is an excellent benchmark for assessing the performance of automata learning tools in this area. The ESM has also been studied in other research projects: Ploeger (2005) modelled the ESM and other related managers and verified properties based on the official specifications of the ESM, and Graaf and van Deursen (2007) have checked the consistency of the behavioural specifications defined in the ESM against the RRRT definitions.

Learning a model of the ESM turned out to be more complicated than expected. The top level UML/RRRT statechart from which the software is generated only has 16 states. However, each of these states contains nested states, and in total there are 70 states that do not have further nested states. Moreover, the C++ code contained in the actions of the transitions also creates some complexity, and this explains why the minimal Mealy machine that models the ESM has 3.410 states. LearnLib has been used to learn models with tens of thousands of states by Raffelt, et al. (2009), and therefore we expected that it would be easy to learn a model for the ESM. However, finding counterexamples for incorrect hypotheses turned out to be challenging due to the large number of 77 inputs. The test algorithms implemented in LearnLib, such as random testing, the W-method by Chow (1978) and Vasilevskii (1973) and the Wp-method by Fujiwara, et al. (1991), failed to deliver counterexamples within an acceptable time. Automata learning techniques have been successfully applied to case studies in which the total number of input symbols is much larger, but in these cases it was possible to reduce the number of inputs to a small number (less than 10) using abstraction techniques (Aarts, et al., 2015 and Howar, et al., 2011). In the case of ESM, use of abstraction techniques only allowed us to reduce the original 156 concrete actions to 77 abstract actions.

We therefore implemented an extension of an algorithm of Lee and Yannakakis (1994) for computing adaptive distinguishing sequences. Even when an adaptive distinguishing sequence does not exist, Lee & Yannakakis' algorithm produces an adaptive sequence that "almost" identifies states. In combination with a standard algorithm for computing separating sequences for pairs of states, we managed to verify states with on average 3 test queries and to learn a model of the ESM with 77 inputs and 3.410 states. We also constructed a model directly from the ESM software and established equivalence with the learned model. To the best of our knowledge, this is the first paper in which active automata learning has been applied to industrial control software. Preliminary evidence suggests that our adaptation of Lee & Yannakakis' algorithm outperforms existing FSM-based conformance algorithms.

During recent years most researchers working on active automata learning focused their efforts on efficient algorithms and tools for the construction of hypothesis models. Our work shows that if we want to further scale automata learning to industrial applications, we also need better algorithms for finding counterexamples for incorrect hypotheses. Following Berg, et al. (2005), our work shows that the context of automata learning provides both new challenges and new opportunities for the application of testing algorithms. All the models for the ESM case study together with the learning and testing statistics are available at http://www.mbsd.cs.ru.nl/publications/papers/fvaan/ESM/, as a benchmark for both the automata learning and testing communities. It is now also included in the automata wiki at http://automata.cs.ru.nl/.

1 *Engine Status Manager*

The focus of this article is the *Engine Status Manager* (*ESM*), a software component that is used to manage the status of the engine of Océ printers and copiers. In this section, the overall structure and context of the ESM will be explained.

1.1 ESRA

The requirements and behaviour of the ESM are defined in a software architecture called Embedded Software Reference Architecture (ESRA). The components defined in this architecture are reused in many of the products developed by Océ and form an important part of these products. This architecture is developed for *cut-sheet* printers or copiers. The term cut-sheet refers to the use of separate sheets of paper as opposed to a *continuous feed* of paper.

An *engine* refers to the printing or scanning part of a printer or copier. Other products can be connected to an engine that pre- or postprocess the paper, for example a cutter, folder, stacker or stapler.

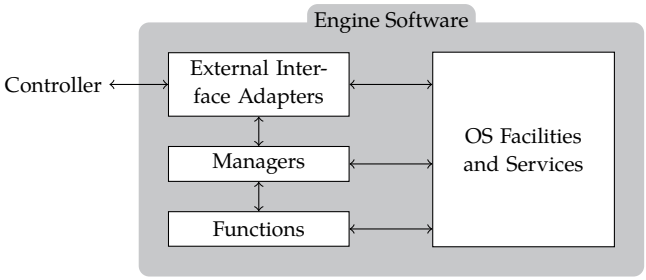


Figure 3.2 Global overview of the engine software.

Figure 3.2 gives an overview of the software in a printer or copier. The *controller* communicates the required actions to the engine software. This includes transport of digital images, status control, print or scan actions and error handling. The controller is responsible for queuing, processing the actions received from the network and operators and delegating the appropriate actions to the engine software. The *managers* communicate with the controller using the *external interface adapters.* These adapters translate the external protocols to internal protocols. The managers manage the different functions of the engine. They are divided by the different functionalities such as status control, print or scan actions or error handling they implement. In order to do this, a manager may communicate with other managers and functions. A *function* is responsible for a specific set of hardware components. It translates commands from the managers to the function hardware and reports the status and other information of the function hardware to the managers. This hardware can for example be the printing hardware or hardware that is not part of the engine hardware such as a stapler. Other functionalities such as logging and debugging are orthogonal to the functions and managers.

1.2 ESM and connected components

The ESM is responsible for the transition from one status of the printer or copier to another. It coordinates the functions to bring them in the correct status. Moreover, it informs all its connected clients (managers or the controller) of status changes. Finally, it handles status transitions when an error occurs.

Figure 3.3 shows the different components to which the ESM is connected. The *Error Handling Manager* (*EHM*), *Action Control Manager* (*ACM*) and other clients request engine statuses. The ESM decides whether a request can be honored immediately, has to be postponed or ignored. If the requested action is processed the ESM requests the functions to go to the appropriate status. The EHM has the highest priority and its requests are processed first. The EHM can request the engine to go into the defect status. The ACM has the next highest priority. The ACM requests the engine to switch between running and standby status. The other clients request transitions between the other statuses, such as idle, sleep, standby and low power. All the other clients have

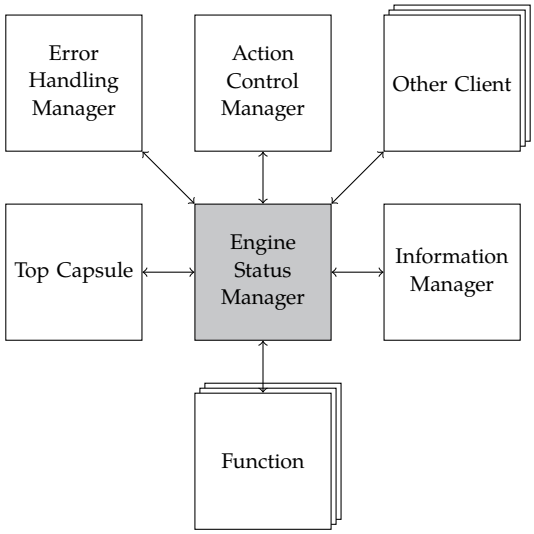


Figure 3.3 Overview of the managers and clients connected to the ESM.

the same lowest priority. The Top Capsule instantiates the ESM and communicates with it during the initialisation of the ESM. The Information Manager provides some parameters during the initialisation.

There are more managers connected to the ESM but they are of less importance and are thus not mentioned here.

1.3 *Rational Rose RealTime*

The ESM has been implemented using *Rational Rose RealTime* (*RRRT*). In this tool so-called *capsules* can be created. Each of these capsules defines a hierarchical *statechart diagram*. Capsules can be connected with each other using *structure diagrams*. Each capsule contains a number of ports that can be connected to ports of other capsules by adding connections in the associated structure diagram. Each of these ports specifies which protocol should be used. This protocol defines which messages may be sent to and from the port. Transitions in the statechart diagram of the capsule can be triggered by arriving messages on a port of the capsule. Messages can be sent to these ports using the action code of the transition. The transitions between the states, actions and guards are defined in C++ code. From the state diagram, C++ source files are generated.

The RRRT language and semantics is based on UML (Object Management Group (OMG), 2004) and ROOM (Selic, et al., 1994). One important concept used in RRRT is the run-to-completion execution model (Eshuis, et al., 2002). This means that when a received message is processed, the execution cannot be interrupted by other arriving messages. These messages are placed in a queue to be processed later.

1.4 The ESM state diagram

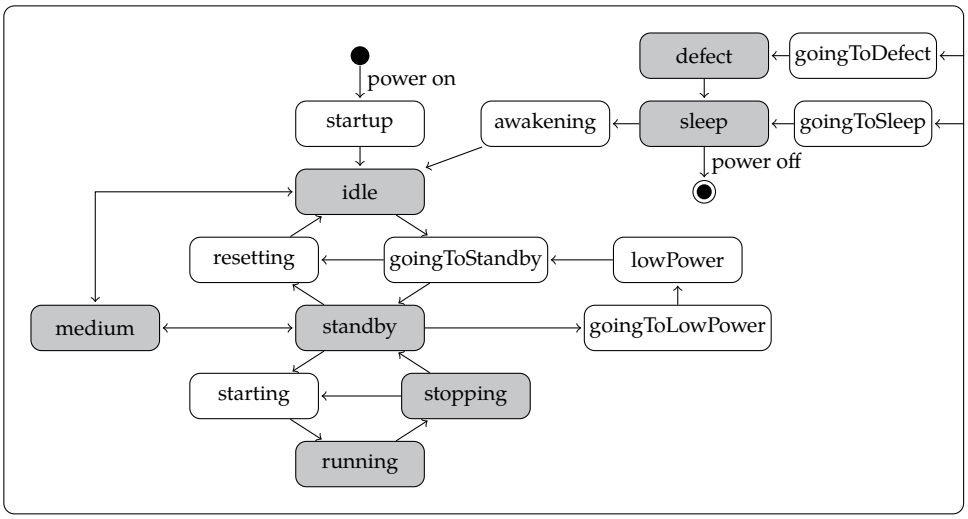


Figure 3.4 Top states and transitions of the ESM.

Figure 3.4 shows the top states of the ESM statechart. The statuses that can be requested by the clients and managers correspond to gray states. The other states are so called *transitory states*. In transitory states the ESM is waiting for the functions to report that they have moved to the corresponding status. Once all functions have reported, the ESM moves to the corresponding status.

The `idle` status indicates that the engine has started up but that it is still *cold* (uncontrolled temperature). The `standby` status indicates that the engine is *warm* and ready for printing or scanning. The `running` status indicates that the engine is printing or scanning. The transitions from the overarching state to the `goingToSleep` and `goingToDefect` states indicate that it is possible to move to the `sleep` or `defect` status from any state. In some cases it is possible to awake from sleep status, in other cases the main power is turned off. The `medium` status is designed for diagnostics. In this status the functions can each be in a different status. For example one function is in standby status while another function is in idle status.

The statechart diagram in Figure 3.4 may seem simple, but it hides many details. Each of the states has up to 5 nested states. In total there are 70 states that do not have further nested states. The C++ code contained in the actions of the transitions is in some cases non-trivial. The possibility to transition from any state to the sleep or defect state also complicates the learning.

2 *Learning the ESM*

In order to learn a model of the ESM, we connected it to LearnLib by Merten, et al. (2011), a state-of-the-art tool for learning Mealy machines developed at the University of Dortmund. A *Mealy machine* is a tuple $M = (I, O, Q, q_0, \delta, \lambda)$, where

- I is a finite set of input symbols,
- O is a finite set of output symbols,
- Q is a finite set of states,
- $q_0 \in Q$ is an initial state,
- $\delta : Q \times I \to Q$ is a transition function, and
- $\lambda : Q \times I \to O$ is an output function.

The behaviour of a Mealy machine is *deterministic*, in the sense that the outputs are fully determined by the inputs. Functions δ and λ are extended to accept sequences in the standard way. We say that Mealy machines $M = (I, O, Q, q_0, \delta, \lambda)$ and $M' = (I', O', Q', q'_0, \delta', \lambda')$ are *equivalent* if they generate an identical sequence of outputs for every sequence of inputs, that is, if $I = I'$ and, for all $w \in I^*$, $\lambda(q_0, w) = \lambda'(q'_0, w)$. If the behaviour of an SUT is described by a Mealy machine M then the task of LearnLib is to learn a Mealy machine M' that is equivalent to M.

2.1 *Experimental set-up*

A clear interface to the ESM has been defined in RRRT. The ESM defines ports from which it receives a predefined set of inputs and to which it can send a predefined set of outputs. However, this interface can only be used within RRRT. In order to communicate with the LearnLib software a TCP connection was set up. An extra capsule was created in RRRT which connects to the ports defined by the ESM. This capsule opened a TCP connection to LearnLib. Inputs and outputs are translated to and from a string format and sent over the connection. Before each membership query, the learner needs to bring the SUT back to its initial state. In other words, LearnLib needs a way to reset the SUT.

Some inputs and outputs sent to and from the ESM carry parameters. These parameters are enumerations of statuses, or integers bounded by the number of functions connected to the ESM. Currently, LearnLib cannot handle inputs with parameters; therefore, we introduced a separate input action for every parameter value. Based on domain knowledge and discussions with the Océ engineers, we could group some of these inputs together and reduce the total number of inputs. When learning the ESM using one function, 83 concrete inputs are grouped into four abstract inputs. When using two functions, 126 concrete inputs can be grouped. When an abstract input needs to be sent to the ESM, one concrete input of the represented group is randomly selected, as in the approach of Aarts, et al. (2015). This is a valid abstraction because all the inputs in the group have exactly the same behaviour in any state of the ESM.

This has been verified by doing code inspection. No other abstractions were found during the research. After the inputs are grouped a total of 77 inputs remain when learning the ESM using 1 function, and 105 inputs remain when using 2 functions.

It was not immediately obvious how to model the ESM by a Mealy machine, since some inputs trigger no output, whereas other inputs trigger several outputs. In order to resolve this, we benefited from the run-to-completion execution model used in RRRT. Whenever an input is sent, all the outputs are collected until quiescence is detected. Next, all the outputs are concatenated and are sent to LearnLib as a single aggregated output. In model-based testing, quiescence is usually detected by waiting for a fixed time-out period. However, this causes the system to be mostly idle while waiting for the time-out, which is inefficient. In order to detect quiescence faster, we exploited the run-to-completion execution model used by RRRT: we modified the ESM to respond to a new low-priority test input with a (single) special output. This test input is sent after each normal input. Only after the normal input is processed and all the generated outputs have been sent, the test input is processed and the special output is generated; upon its reception, quiescence can be detected immediately and reliably.

2.2 *Test selection strategies*

In the ESM case study the most challenging problem was finding counterexamples for the hypotheses constructed during learning.

LearnLib implements several algorithms for conformance testing, one of which is a random walk algorithm. The random walk algorithm works by first selecting the length of the test query according to a geometric distribution, cut off at a fixed upper bound. Each of the input symbols in the test query is then randomly selected from the input alphabet I from a uniform distribution. In order to find counterexamples, a specific sequence of input symbols is needed to arrive at the state in the SUT that differentiates it from the hypothesis. The upper bound for the size of this search space is $|I|^n$ where $|I|$ is the size of the input alphabet used, and n the length of the counterexample that needs to be found. If this sequence is long the chance of finding it is small. Because the ESM has many different input symbols to choose from, finding the correct one is hard. When learning the ESM with 1 function there are 77 possible input symbols. If for example the length of the counterexample needs to be at least 6 inputs to identify a certain state, then the upper bound on the number of test queries would be around 2×10^{11}. An average test query takes around 1 ms, so it would take about 7 years to execute these test queries.

Augmented DS-method[15]**.** In order to reduce the number of tests, Chow (1978) and Vasilevskii (1973) pioneered the so called W-method. In their framework a test query

[15] This was later called the *hybrid ADS-method*.

consists of a prefix p bringing the SUT to a specific state, a (random) middle part m and a suffix s assuring that the SUT is in the appropriate state. This results in a test suite of the form $PI^{\leqslant k}W$, where P is a set of (shortest) access sequences, $I^{\leqslant k}$ the set of all sequences of length at most k, and W is a characterisation set. Classically, this characterisation set is constructed by taking the set of all (pairwise) separating sequences. For $k = 1$ this test suite is complete in the sense that if the SUT passes all tests, then either the SUT is equivalent to the specification or the SUT has strictly more states than the specification. By increasing k we can check additional states.

We tried using the W-method as implemented by LearnLib to find counterexamples. The generated test suite, however, was still too big in our learning context. Fujiwara, et al. (1991) observed that it is possible to let the set W depend on the state the SUT is supposed to be. This allows us to only take a subset of W which is relevant for a specific state. This slightly reduces the test suite without losing the power of the full test suite. This method is known as the Wp-method. More importantly, this observation allows for generalisations where we can carefully pick the suffixes.

In the presence of an (adaptive) distinguishing sequence one can take W to be a single suffix, greatly reducing the test suite. Lee and Yannakakis (1994) describe an algorithm (which we will refer to as the LY algorithm) to efficiently construct this sequence, if it exists. In our case, unfortunately, most hypotheses did not enjoy existence of an adaptive distinguishing sequence. In these cases the incomplete result from the LY algorithm still contained a lot of information which we augmented by pairwise separating sequences.

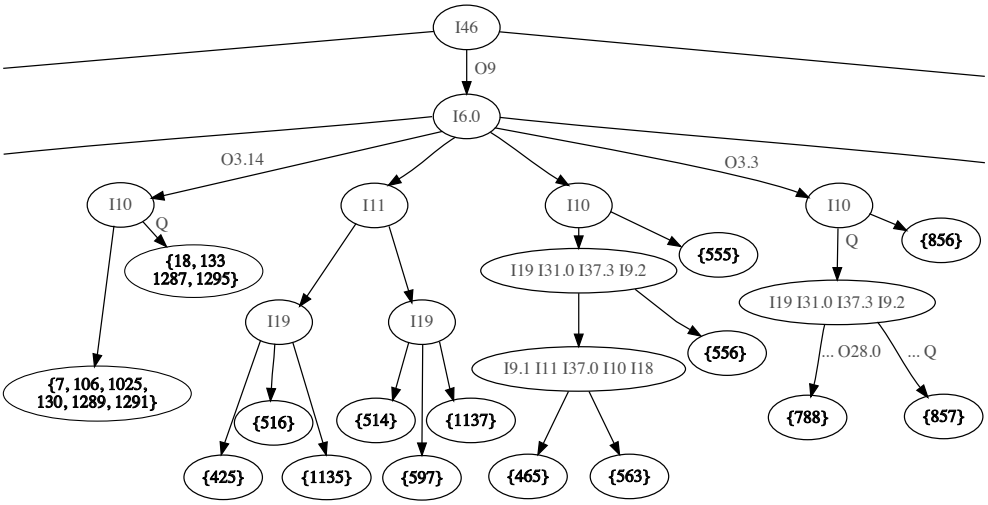


Figure 3.5 A small part of an incomplete distinguishing sequence as produced by the LY algorithm. Leaves contain a set of possible initial states, inner nodes have input sequences and edges correspond to different output symbols (of which we only drew some), where Q stands for quiescence.

As an example we show an incomplete adaptive distinguishing sequence for one of the hypothesis in Figure 3.5. When we apply the input sequence I46 I6.0 I10 I19 I31.0 I37.3 I9.2 and observe outputs O9 O3.3 Q ... O28.0, we know for sure that the SUT was in state 788. Unfortunately, not all paths lead to a singleton set. When for instance we apply the sequence I46 I6.0 I10 and observe the outputs O9 O3.14 Q, we know for sure that the SUT was in one of the states 18, 133, 1287 or 1295. In these cases we have to perform more experiments and we resort to pairwise separating sequences.

We note that this augmented DS-method is in the worst case not any better than the classical Wp-method. In our case, however, it greatly reduced the test suites.

Once we have our set of suffixes, which we call Z now, our test algorithm works as follows. The algorithm first exhausts the set $PI^{\leqslant 1}Z$. If this does not provide a counterexample, we will randomly pick test queries from PI^2I^*Z, where the algorithm samples uniformly from P, I^2 and Z (if Z contains more that 1 sequence for the supposed state) and with a geometric distribution on I^*.

Sub-alphabet selection. Using the above method the algorithm still failed to learn the ESM. By looking at the RRRT-based model we were able to see why the algorithm failed to learn. In the initialisation phase, the controller gives exceptional behaviour when providing a certain input eight times consecutively. Of course such a sequence is hard to find in the above testing method. With this knowledge we could construct a single counterexample by hand by which means the algorithm was able to learn the ESM.

In order to automate this process, we defined a sub-alphabet of actions that are important during the initialisation phase of the controller. This sub-alphabet will be used a bit more often than the full alphabet. We do this as follows. We start testing with the alphabet which provided a counterexample for the previous hypothesis (for the first hypothesis we take the sub-alphabet). If no counterexample can be found within a specified query bound, then we repeat with the next alphabet. If both alphabets do not produce a counterexample within the bound, the bound is increased by some factor and we repeat all. This method only marginally increases the number of tests. But it did find the right counterexample we first had to construct by hand.

2.3 *Results*

Using the learning set-up discussed in Section 2.1 and the test selection strategies discussed in Section 2.2, a model of the ESM using 1 function could be learned. After an additional eight hours of testing no counterexample was found and the experiment was stopped. The following list gives the most important statistics gathered during the learning:

- The learned model has 3.410 states.
- Altogether, 114 hypotheses were generated.
- The time needed for learning the final hypothesis was 8 h, 26 min, and 19 s.

– 29.933.643 membership queries were posed (on average 35,77 inputs per query).
– 30.629.711 test queries were required (on average 29,06 inputs per query).

3 *Verification*

To verify the correctness of the model that was learned using LearnLib, we checked its equivalence with a model that was generated directly from the code.

3.1 *Approach*

As mentioned already, the ESM has been implemented using Rational Rose RealTime (RRRT). Thus a statechart representation of the ESM is available. However, we have not been able to find a tool that translates RRRT models to Mealy machines, allowing us to compare the RRRT-based model of the ESM with the learned model. We considered several formalisms and tools that were proposed in the literature to flatten statecharts to state machines. The first one was a tool for hierarchical timed automata (HTA) by David, et al. (2002). However, we found it hard to translate the output of this tool, a network of Uppaal timed automata, to a Mealy machine that could be compared to the learned model. The second tool that we considered has been developed by Hansen, et al. (2010). This tool misses some essential features, for example the ability to assign new values to state variables on transitions. Finally, we considered a formalism called object-oriented action systems (OOAS) by Krenn, et al. (2009), but no tools to use this could be found.

In the end we decided to implement the required model transformations ourselves. Figure 3.6 displays the different formats for representing models that we used and the transformations between those formats.

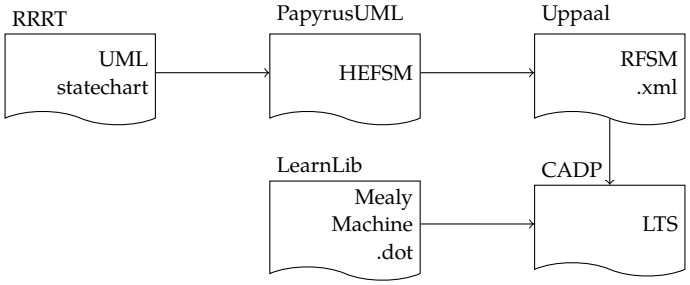


Figure 3.6 Formats for representing models and transformations between formats.

We used the bisimulation checker of CADP by Garavel, et al. (2011) to check the equivalence of labelled transition system models in `.aut` format. The Mealy machine models learned by LearnLib are represented as `.dot` files. A small script converts

these Mealy machines to labelled transition systems in `.aut` format. We used the Uppaal tool by Behrmann, et al. (2006) as an editor for defining extended finite state machines (EFSM), represented as `.xml` files. A script developed in the ITALIA project (`http://www.italia.cs.ru.nl/`) converts these EFSM models to LOTOS, and then CADP takes care of the conversion from LOTOS to the `.aut` format.

The Uppaal syntax is not sufficiently expressive to directly encode the RRRT definition of the ESM, since this definition makes heavy use of UML (Object Management Group (OMG), 2004) concepts such as state hierarchy and transitions from composite states, concepts which are not present in Uppaal. Using Uppaal would force us to duplicate many transitions and states.

We decided to manually create an intermediate hierarchical EFSM (HEFSM) model using the UML drawing tool PapyrusUML (Lanusse, et al., 2009). The HEFSM model closely resembles the RRRT UML model, but many elements used in UML state machines are left out because they are not needed for modelling the ESM and complicate the transformation process.

3.2 *Model transformations*

We explain the transformation from the HEFSM model to the EFSM model using examples. The transformation is divided into five steps, which are executed in order:

1. combine transitions without input or output signal,
2. transform supertransitions,
3. transform internal transitions,
4. add input signals that do not generate an output, and
5. replace invocations of the next function.

1. Empty transitions. In order to make the model more readable and to make it easy to model `if` and `switch` statements in the C++ code the HEFSM model allows for transitions without a signal. These transitions are called *empty* transitions. An empty transition can still contain a guard and an assignment. However these kinds of transitions are only allowed on states that only contain empty outgoing transitions. This was done to make the transformation easy and the model easy to read.

In order to transform a state with empty transitions all the incoming and outgoing transitions are collected. For each combination of incoming transition a and outgoing transition b a new transition c is created with the source of a as source and the target of b as target. The guard for transition c evaluates to true if and only if the guard of a and b both evaluate to true. The assignment of c is the concatenation of the assignment of a and b. The signal of c will be the signal of a because b cannot have a signal. Once all the new transitions are created all the states with empty transitions are removed together with all their incoming and outgoing transitions.

Figure 3.7 shows an example model with empty transitions and its transformed version. Each of the incoming transitions from the state `B` is combined with each of

the outgoing transitions. This results into two new transitions. The old transitions and state B are removed.

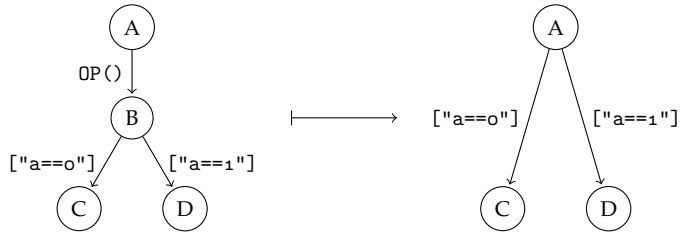


Figure 3.7 Example of empty transition transformation. On the left the original version. On the right the transformed version.

2. Supertransitions. The RRRT model of the ESM contains many transitions originating from a composite state. Informally, these *supertransitions* can be taken in each of the substates of the composite state if the guard evaluates to true. In order to model the ESM as closely as possible, supertransitions are also supported in the HEFSM model.

In RRRT transitions are evaluated from bottom to top. This means that first the transitions from the leaf state are considered, then transitions from its parent state and then from its parent's parent state, etc. Once a transition for which the guard evaluates to true and the correct signal has been found it is taken. When flattening the statechart, we modified the guards of supertransitions to ensure the correct priorities.

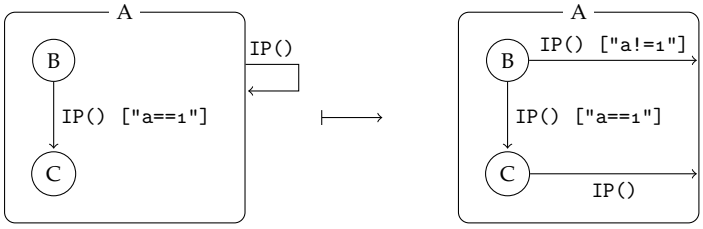


Figure 3.8 Example of supertransition transformation. On the left the original version. On the right the transformed version.

Figure 3.8 shows an example model with supertransitions and its transformed version. The supertransition from state A can be taken at each of A's leaf states B and C. The transformation removes the original supertransition and creates a new transition at states B and C using the same target state. For leaf state C this is easy because it does not contain a transition with the input signal IP. In state B the transition to state C would be taken if a signal IP was processed and the state variable a equals 1. The supertransition can only be taken if the other transition cannot be taken. This is why the negation of other the guard is added to the new transition. If the original supertransition is an

internal transition the model needs further transformation after this transformation. This is described in the next paragraph. If the original supertransition is not an internal transition the new transitions will have the initial state of A as target.

3. Internal transitions. The ESM model also makes use of *internal transitions* in RRRT. Using such a transition the current state does not change. If such a transition is defined on a composite state it can be taken from all of the substates and return to the same leaf state it originated from. If defined on a composite state it is thus also a supertransition. This is also possible in the HEFSM model. In order to transform an internal transition it is first seen as a supertransition and the above transformation is applied. Then the target of the transition is simply set to the leaf state it originates from. An example can be seen in Figure 3.8. If the supertransition from state A is also defined to be an internal transition the transformed version on the right would need another transformation. The new transitions that now have the target state A would be transformed to have the same target state as their current source state.

4. Quiescent transitions. In order to reduce the number of transitions in the HEFSM model quiescent transitions are added automatically. For every state all the transitions for each signal are collected in a set T. A new self transition α is added for each signal. The guard for transition α evaluates to true if and only if none of the guards of the transactions in T evaluates to true. This makes the HEFSM input enabled without having to specify all the transitions.

5. The next function. In RRRT it is possible to write the guard and assignment in C++ code. It is thus possible that the value of a variable changes while an input signal is processed. In the HEFSM however all the assignments only take effect after the input signal is processed. In order to simulate this behaviour the *next* function is used. This function takes a variable name and evaluates to the value of this variable after the transition.

3.3 *Results*

Figure 3.9 shows a visualisation of the learned model that was generated using Gephi (Bastian, et al., 2009). States are coloured according to the strongly connected components. The number of transitions between two states is represented by the thickness of the edge. The large number of states (3.410) and transitions (262.570) makes it hard to visualise this model. Nevertheless, the visualisation does provide insight in the behaviour of the ESM. The three protrusions at the bottom of Figure 3.9 correspond to deadlocks in the model. These deadlocks are "error" states that are present in the ESM by design. According to the Océ engineers, the sequences of inputs that are needed to drive the ESM into these deadlock states will always be followed by a system power reset. The protrusion at the top right of the figure corresponds to the initialisation phase of the ESM. This phase is performed only once and thus only transitions from the initialisation cluster to the main body of states are present.

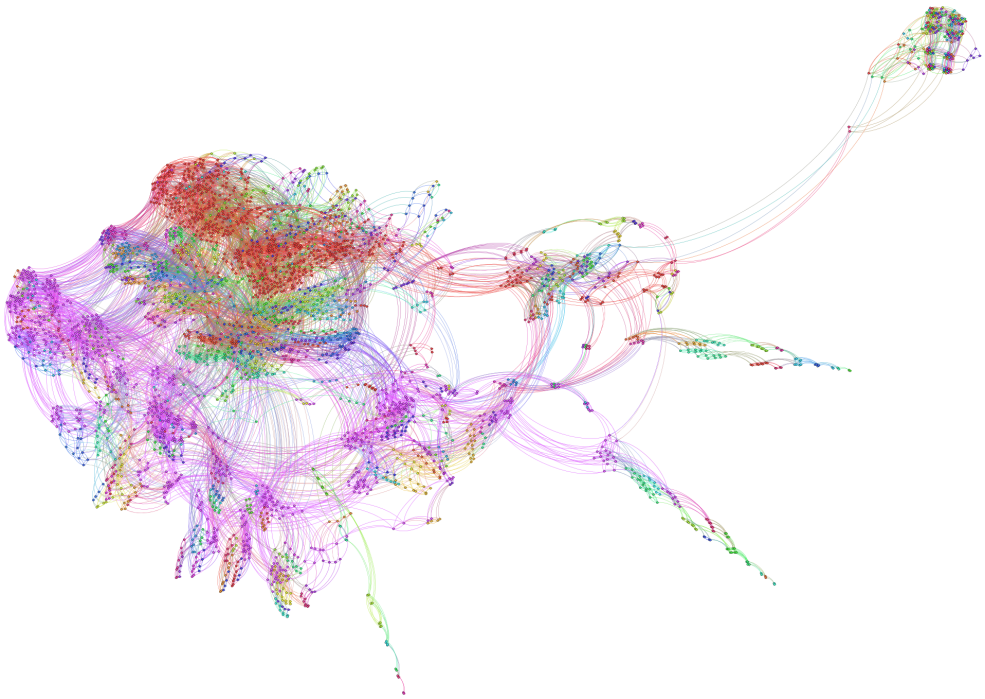


Figure 3.9 Final model of the ESM.

During the construction of the RRRT-based model, the ESM code was thoroughly inspected. This resulted in the discovery of missing behaviour in one transition of the ESM code. An Océ software engineer confirmed that this behaviour is a (minor) bug, which will be fixed. We have verified the equivalence of the learned model and the RRRT-based model by using CADP (Garavel, et al., 2011).

4 *Conclusions and Future Work*

Using an extension of algorithm by Lee and Yannakakis (1994) for adaptive distinguishing sequences, we succeeded to learn a Mealy machine model of a piece of widely used industrial control software. Our extension of Lee & Yannakakis' algorithm is rather obvious, but nevertheless appears to be new. Preliminary evidence suggests that it outperforms existing conformance testing algorithms. We are currently performing experiments in which we compare the new algorithm with other test algorithms on a number of realistic benchmarks.

There are several possibilities for extending the ESM case study. To begin with, one could try to learn a model of the ESM with more than one function. Another interesting possibility would be to learn models of the EHM, ACM, and other managers connected to the ESM. Using these models some of the properties discussed by Ploeger

(2005) could be verified at a more detailed level. We expect that the combination of LearnLib with the extended Lee & Yannakakis algorithm can be applied to learn models of many other software components.

In the specific case study described in this article, we know that our learning algorithm has succeeded to learn the correct model, since we established equivalence with a reference model that was constructed independently from the RRRT model of the ESM software. In the absence of a reference model, we can never guarantee that the actual system behaviour conforms to a learned model. In order to deal with this problem, it is important to define metrics that quantify the difference (or distance) between a hypothesis and a correct model of the SUT and to develop test generation algorithms that guarantee an upper bound on this difference. Preliminary work in this area is reported by Smetsers, et al. (2014).

Acknowledgements

We thank Lou Somers for suggesting the ESM case study and for his support of our research. Fides Aarts and Harco Kuppens helped us with the use of LearnLib and CADP, and Jan Tretmans gave useful feedback.

Chapter 4
Minimal Separating Sequences for All Pairs of States

Rick Smetsers
Radboud University

Joshua Moerman
Radboud University

David N. Jansen
Radboud University

Abstract

Finding minimal separating sequences for all pairs of inequivalent states in a finite state machine is a classic problem in automata theory. Sets of minimal separating sequences, for instance, play a central role in many conformance testing methods. Moore has already outlined a partition refinement algorithm that constructs such a set of sequences in $\mathcal{O}(mn)$ time, where m is the number of transitions and n is the number of states. In this chapter, we present an improved algorithm based on the minimisation algorithm of Hopcroft that runs in $\mathcal{O}(m \log n)$ time. The efficiency of our algorithm is empirically verified and compared to the traditional algorithm.

This chapter is based on the following publication:
Smetsers, R., Moerman, J., & Jansen, D. N. (2016). Minimal Separating Sequences for All Pairs of States. In *Language and Automata Theory and Applications - 10th International Conference, LATA, Proceedings*. Springer. doi:10.1007/978-3-319-30000-9_14

In diverse areas of computer science and engineering, systems can be modelled by *finite state machines* (FSMs). One of the cornerstones of automata theory is minimisation of such machines – and many variation thereof. In this process one obtains an equivalent minimal FSM, where states are different if and only if they have different behaviour. The first to develop an algorithm for minimisation was Moore (1956). His algorithm has a time complexity of $\mathcal{O}(mn)$, where m is the number of transitions, and n is the number of states of the FSM. Later, Hopcroft (1971) improved this bound to $\mathcal{O}(m \log n)$.

Minimisation algorithms can be used as a framework for deriving a set of *separating sequences* that show *why* states are inequivalent. The separating sequences in Moore's framework are of minimal length (Gill, 1962). Obtaining minimal separating sequences in Hopcroft's framework, however, is a non-trivial task. In this chapter, we present an algorithm for finding such minimal separating sequences for all pairs of inequivalent states of a FSM in $\mathcal{O}(m \log n)$ time.

Coincidentally, Bonchi and Pous (2013) recently introduced a new algorithm for the equally fundamental problem of proving equivalence of states in non-deterministic automata. As both their and our work demonstrate, even classical problems in automata theory can still offer surprising research opportunities. Moreover, new ideas for well-studied problems may lead to algorithmic improvements that are of practical importance in a variety of applications.

One such application for our work is in *conformance testing*. Here, the goal is to test if a black box implementation of a system is functioning as described by a given FSM. It consists of applying sequences of inputs to the implementation, and comparing the output of the system to the output prescribed by the FSM. Minimal separating sequences are used in many test generation methods (Dorofeeva, et al., 2010). Therefore, our algorithm can be used to improve these methods.

1 Preliminaries

We define a *FSM* as a Mealy machine $M = (I, O, S, \delta, \lambda)$, where I, O and S are finite sets of *inputs*, *outputs* and *states* respectively, $\delta\colon S \times I \to S$ is a *transition function* and $\lambda\colon S \times I \to O$ is an *output function*. The functions δ and λ are naturally extended to $\delta\colon S \times I^* \to S$ and $\lambda\colon S \times I^* \to O^*$. Moreover, given a set of states $S' \subseteq S$ and a sequence $x \in I^*$, we define $\delta(S', x) = \{\delta(s, x) \mid s \in S'\}$ and $\lambda(S', x) = \{\lambda(s, x) \mid s \in S'\}$. The *inverse transition function* $\delta^{-1}\colon S \times I \to \mathcal{P}(S)$ is defined as $\delta^{-1}(s, a) = \{t \in S \mid \delta(t, a) = s\}$.

Observe that Mealy machines are deterministic and input-enabled (i.e., complete) by definition. The initial state is not specified because it is of no importance in what follows. For the remainder of this chapter we fix a machine $M = (I, O, S, \delta, \lambda)$. We use n to denote its number of states, that is $n = |S|$, and m to denote its number of transitions, that is $m = |S| \times |I|$.

Definition 1. States s and t are *equivalent* if $\lambda(s, x) = \lambda(t, x)$ for all x in I^*.

We are interested in the case where s and t are not equivalent, i.e., *inequivalent*. If all pairs of distinct states of a machine M are inequivalent, then M is *minimal*. An example of a minimal FSM is given in Figure 4.1.

Definition 2. A *separating sequence* for states s and t in s is a sequence $x \in I^*$ such that $\lambda(s, x) \neq \lambda(t, x)$. We say x is *minimal* if $|y| \geqslant |x|$ for all separating sequences y for s and t.

A separating sequence always exists if two states are inequivalent, and there might be multiple minimal separating sequences. Our goal is to obtain minimal separating sequences for all pairs of inequivalent states of M.

1.1 Partition Refinement

In this section we will discuss the basics of minimisation. Both Moore's algorithm and Hopcroft's algorithm work by means of partition refinement. A similar treatment (for DFAs) is given by Gries (1973).

A *partition* P of S is a set of pairwise disjoint non-empty subsets of S whose union is exactly S. Elements in P are called *blocks*. If P and P' are partitions of S, then P' is a *refinement* of P if every block of P' is contained in a block of P. A partition refinement algorithm constructs the finest partition under some constraint. In our context the constraint is that equivalent states belong to the same block.

Definition 3. A partition is *valid* if equivalent states are in the same block.

Partition refinement algorithms for FSMs start with the trivial partition $P = \{S\}$, and iteratively refine P until it is the finest valid partition (where all states in a block are equivalent). The blocks of such a *complete* partition form the states of the minimised FSM, whose transition and output functions are well-defined because states in the same block are equivalent.

Let B be a block and a be an input. There are two possible reasons to split B (and hence refine the partition). First, we can *split* B *with respect to output after* a if the set $\lambda(B, a)$ contains more than one output. Second, we can *split* B *with respect to the state after* a if there is no single block B' containing the set $\delta(B, a)$. In both cases it is obvious what the new blocks are: in the first case each output in $\lambda(B, a)$ defines a new block, in the second case each block containing a state in $\delta(B, a)$ defines a new block. Both types of refinement preserve validity.

Partition refinement algorithms for FSMs first perform splits w.r.t. output, until there are no such splits to be performed. This is precisely the case when the partition is *acceptable*.

Definition 4. A partition is *acceptable* if for all pairs s, t of states contained in the same block and for all inputs a in I, $\lambda(s, a) = \lambda(t, a)$.

Any refinement of an acceptable partition is again acceptable. The algorithm continues performing splits w.r.t. state, until no such splits can be performed. This is exactly the case when the partition is stable.

Definition 5. A partition is *stable* if it is acceptable and for any input a in I and states s and t that are in the same block, states $\delta(s, a)$ and $\delta(t, a)$ are also in the same block.

Since an FSM has only finitely many states, partition refinement will terminate. The output is the finest valid partition which is acceptable and stable. For a more formal treatment on partition refinement we refer to Gries (1973).

1.2 *Splitting Trees and Refinable Partitions*

Both types of splits described above can be used to construct a separating sequence for the states that are split. In a split w.r.t. the output after a, this sequence is simply a. In a split w.r.t. the state after a, the sequence starts with an a and continues with the separating sequence for states in $\delta(B, a)$. In order to systematically keep track of this information, we maintain a *splitting tree*. The splitting tree was introduced by Lee and Yannakakis (1994) as a data structure for maintaining the operational history of a partition refinement algorithm.

Definition 6. A *splitting tree* for M is a rooted tree T with a finite set of nodes with the following properties:

- Each node u in T is labelled by a subset of S, denoted $l(u)$.
- The root is labelled by S.
- For each inner node u, $l(u)$ is partitioned by the labels of its children.
- Each inner node u is associated with a sequence $\sigma(u)$ that separates states contained in different children of u.

We use $C(u)$ to denote the set of children of a node u. The *lowest common ancestor* (lca) for a set $S' \subseteq S$ is the node u such that $S' \subseteq l(u)$ and $S' \not\subseteq l(v)$ for all $v \in C(u)$ and is denoted by $\mathrm{lca}(S')$. For a pair of states s and t we use the shorthand $\mathrm{lca}(s, t)$ for $\mathrm{lca}(\{s, t\})$.

The labels $l(u)$ can be stored as a *refinable partition* data structure (Valmari & Lehtinen, 2008). This is an array containing a permutation of the states, ordered so that states in the same block are adjacent. The label $l(u)$ of a node then can be indicated by a slice of this array. If node u is split, some states in the *slice* $l(u)$ may be moved to create the labels of its children, but this will not change the *set* $l(u)$.

A splitting tree T can be used to record the history of a partition refinement algorithm because at any time the leaves of T define a partition on S, denoted $P(T)$. We say a splitting tree T is valid (resp. acceptable, stable, complete) if $P(T)$ is as such. A leaf can be expanded in one of two ways, corresponding to the two ways a block can be split. Given a leaf u and its block $B = l(u)$ we define the following two splits:

(**split-output**) Suppose there is an input a such that B can be split w.r.t output after a. Then we set $\sigma(u) = a$, and we create a node for each subset of B that produces the same output x on a. These nodes are set to be children of u.

(**split-state**) Suppose there is an input a such that B can be split w.r.t. the state after a. Then instead of splitting B as described before, we proceed as follows. First, we locate the node $v = lca(\delta(B, a))$. Since v cannot be a leaf, it has at least two children whose labels contain elements of $\delta(B, a)$. We can use this information to expand the tree as follows. For each node w in $C(v)$ we create a child of u labelled $\{s \in B \mid \delta(s, a) \in l(w)\}$ if the label contains at least one state. Finally, we set $\sigma(u) = a\sigma(v)$.

A straight-forward adaptation of partition refinement for constructing a stable splitting tree for M is shown in Algorithm 4.1. The termination and the correctness of the algorithm outlined in Section 1.1 are preserved. It follows directly that states are equivalent if and only if they are in the same label of a leaf node.

Require: An FSM M
Ensure: A valid and stable splitting tree T
1 initialise T to be a tree with a single node labelled S
2 **repeat**
3 find $a \in I, B \in P(T)$ such that we can split B w.r.t. output $\lambda(\cdot, a)$
4 expand the $u \in T$ with $l(u) = B$ as described in (split-output)
5 **until** $P(T)$ is acceptable
6 **repeat**
7 find $a \in I, B \in P(T)$ such that we can split B w.r.t. state $\delta(\cdot, a)$
8 expand the $u \in T$ with $l(u) = B$ as described in (split-state)
9 **until** $P(T)$ is stable

Algorithm 4.1 Constructing a stable splitting tree.

Example 7. Figure 4.1 shows an FSM and a complete splitting tree for it. This tree is constructed by Algorithm 4.1 as follows. First, the root node is labelled by $\{s_0, ..., s_5\}$. The even and uneven states produce different outputs after a, hence the root node is split. Then we note that s_4 produces a different output after b than s_0 and s_2, so $\{s_0, s_2, s_4\}$ is split as well. At this point T is acceptable: no more leaves can be split w.r.t. output. Now, the states $\delta(\{s_1, s_3, s_5\}, a)$ are contained in different leaves of T. Therefore, $\{s_1, s_3, s_5\}$ is split into $\{s_1, s_5\}$ and $\{s_3\}$ and associated with sequence ab. At this point, $\delta(\{s_0, s_2\}, a)$ contains states that are in both children of $\{s_1, s_3, s_5\}$, so $\{s_0, s_2\}$ is split and the associated sequence is aab. We continue until T is complete.

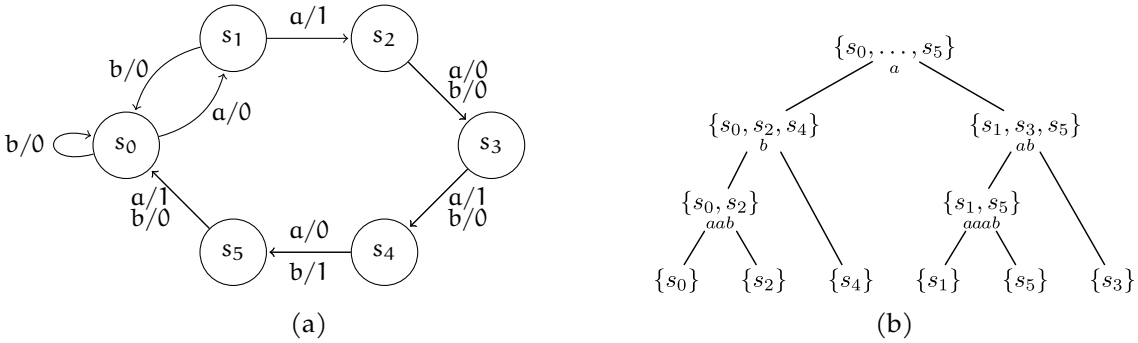


Figure 4.1 An FSM (a) and a complete splitting tree for it (b).

2 *Minimal Separating Sequences*

In Section 1.2 we have described an algorithm for constructing a complete splitting tree. This algorithm is non-deterministic, as there is no prescribed order on the splits. In this section we order them to obtain minimal separating sequences.

Let u be a non-root inner node in a splitting tree, then the sequence $\sigma(u)$ can also be used to split the parent of u. This allows us to construct splitting trees where children will never have shorter sequences than their parents, as we can always split with those sequences first. Trees obtained in this way are guaranteed to be *layered*, which means that for all nodes u and all $u' \in C(u)$, $|\sigma(u)| \leqslant |\sigma(u')|$. Each layer consists of nodes for which the associated separating sequences have the same length.

Our approach for constructing minimal sequences is to ensure that each layer is as large as possible before continuing to the next one. This idea is expressed formally by the following definitions.

Definition 8. A splitting tree T is k-*stable* if for all states s and t in the same leaf we have $\lambda(s, x) = \lambda(t, x)$ for all $x \in I^{\leqslant k}$.

Definition 9. A splitting tree T is *minimal* if for all states s and t in different leaves $\lambda(s, x) \neq \lambda(t, x)$ implies $|x| \geqslant |\sigma(\mathrm{lca}(s, t))|$ for all $x \in I^*$.

Minimality of a splitting tree can be used to obtain minimal separating sequences for pairs of states. If the tree is in addition stable, we obtain minimal separating sequences for all inequivalent pairs of states. Note that if a minimal splitting tree is $(n-1)$-stable (n is the number of states of M), then it is stable (Definition 5). This follows from the well-known fact that $n-1$ is an upper bound for the length of a minimal separating sequence (Moore, 1956).

Algorithm 4.2 ensures a stable and minimal splitting tree. The first repeat-loop is the same as before (in Algorithm 4.1). Clearly, we obtain a 1-stable and minimal splitting tree here. It remains to show that we can extend this to a stable and minimal

splitting tree. Algorithm 4.3 will perform precisely one such step towards stability, while maintaining minimality. Termination follows from the same reason as for Algorithm 4.1. Correctness for this algorithm is shown by the following key lemma. We will denote the input tree by T and the tree after performing Algorithm 4.3 by T'. Observe that T is an initial segment of T'.

Lemma 10. Algorithm 4.3 ensures a $(k+1)$-stable minimal splitting tree.

Proof. Let us proof stability. Let s and t be in the same leaf of T' and let $x \in I^*$ be such that $\lambda(s,x) \neq \lambda(t,x)$. We show that $|x| > k+1$.

Suppose for the sake of contradiction that $|x| \leqslant k+1$. Let u be the leaf containing s and t and write $x = ax'$. We see that $\delta(s,a)$ and $\delta(t,a)$ are separated by k-stability of T. So the node $v = \mathrm{lca}(\delta(l(u),a))$ has children and an associated sequence $\sigma(v)$. There are two cases:

- $|\sigma(v)| < k$, then $a\sigma(v)$ separates s and t and is of length $\leqslant k$. This case contradicts the k-stability of T.
- $|\sigma(v)| = k$, then the loop in Algorithm 4.3 will consider this case and split. Note that this may not split s and t (it may occur that $a\sigma(v)$ splits different elements in $l(u)$). We can repeat the above argument inductively for the newly created leaf containing s and t. By finiteness of $l(u)$, the induction will stop and, in the end, s and t are split.

Both cases end in contradiction, so we conclude that $|x| > k+1$.

Let us now prove minimality. It suffices to consider only newly split states in T'. Let s and t be two states with $|\sigma(\mathrm{lca}(s,t))| = k+1$. Let $x \in I^*$ be a sequence such that $\lambda(s,x) \neq \lambda(t,x)$. We need to show that $|x| \geqslant k+1$. Since $x \neq \epsilon$ we can write $x = ax'$ and consider the states $s' = \delta(s,a)$ and $t' = \delta(t,a)$ which are separated by x'. Two things can happen:

- The states s' and t' are in the same leaf in T. Then by k-stability of T we get $\lambda(s',y) = \lambda(t',y)$ for all $y \in I^{\leqslant k}$. So $|x'| > k$.
- The states s' and t' are in different leaves in T and let $u = \mathrm{lca}(s',t')$. Then $a\sigma(u)$ separates s and t. Since s and t are in the same leaf in T we get $|a\sigma(u)| \geqslant k+1$ by k-stability. This means that $|\sigma(u)| \geqslant k$ and by minimality of T we get $|x'| \geqslant k$.

In both cases we have shown that $|x| \geqslant k+1$ as required. □

Example 11. Figure 4.2a shows a stable and minimal splitting tree T for the machine in Figure 4.1. This tree is constructed by Algorithm 4.2 as follows. It executes the same as Algorithm 4.1 until we consider the node labelled $\{s_0, s_2\}$. At this point $k = 1$. We observe that the sequence of $\mathrm{lca}(\delta(\{s_0,s_2\},a))$ has length 2, which is too long, so we continue with the next input. We find that we can indeed split w.r.t. the state after b, so the associated sequence is ba. Continuing, we obtain the same partition as before, but with smaller witnesses.

The internal data structure (a refinable partition) is shown in Figure 4.2(b): the array with the permutation of the states is at the bottom, and every block includes

Require: An FSM M with n states
Ensure: A stable, minimal splitting tree T
1 initialise T to be a tree with a single node labelled S
2 **repeat**
3 find $a \in I, B \in P(T)$ such that we can split B w.r.t. output $\lambda(\cdot, a)$
4 expand the $u \in T$ with $l(u) = B$ as described in (split-output)
5 **until** $P(T)$ is acceptable
6 **for** $k = 1$ **to** $n - 1$ **do**
7 invoke Algorithm 4.3 or Algorithm 4.4 on T for k
8 **end for**

Algorithm 4.2 Constructing a stable and minimal splitting tree.

Require: A k-stable and minimal splitting tree T
Ensure: T is a $(k+1)$-stable, minimal splitting tree
1 **for all** leaves $u \in T$ and all inputs $a \in I$ **do**
2 $v \leftarrow \mathrm{lca}(\delta(l(u), a))$
3 **if** v is an inner node and $|\sigma(v)| = k$ **then**
4 expand u as described in (split-state) (this generates new leaves)
5 **end if**
6 **end for**

Algorithm 4.3 A step towards the stability of a splitting tree.

an indication of the slice containing its label and a pointer to its parent (as our final algorithm needs to find the parent block, but never the child blocks).

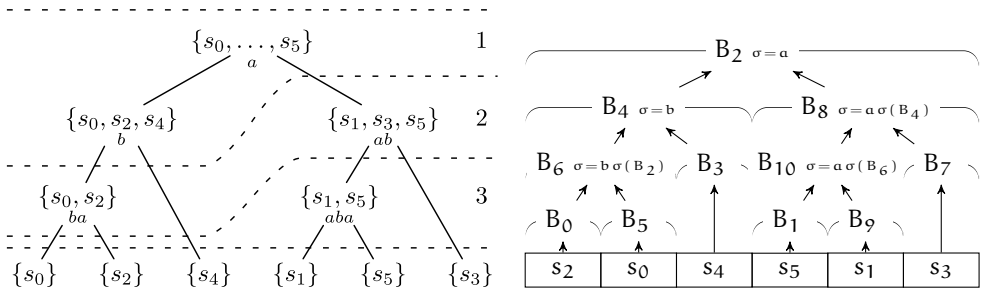


Figure 4.2 (a) A complete and minimal splitting tree for the FSM in Figure 4.1 and (b) its internal refinable partition data structure.

3 Optimising the Algorithm

In this section, we present an improvement on Algorithm 4.3 that uses two ideas described by Hopcroft (1971) in his seminal paper on minimising finite automata: *using the inverse transition set*, and *processing the smaller half*. The algorithm that we present is a drop-in replacement, so that Algorithm 4.2 stays the same except for some bookkeeping. This way, we can establish correctness of the new algorithms more easily. The variant presented in this section reduces the amount of redundant computations that were made in Algorithm 4.3.

Using Hopcroft's first idea, we turn our algorithm upside down: instead of searching for the lca for each leaf, we search for the leaves u for which $l(u) \subseteq \delta^{-1}(l(v), a)$, for each potential lca v and input a. To keep the order of splits as before, we define k-*candidates*.

Definition 12. A k-*candidate* is a node v with $|\sigma(v)| = k$.

A k-candidate v and an input a can be used to split a leaf u if $v = \operatorname{lca}(\delta(l(u), a))$, because in this case there are at least two states s, t in $l(u)$ such that $\delta(s, a)$ and $\delta(t, a)$ are in labels of different nodes in $C(v)$. Refining u this way is called *splitting* u *with respect to* (v, a). The set $C(u)$ is constructed according to (split-state), where each child $w \in C(v)$ defines a child u_w of u with states

$$\begin{aligned} l(u_w) &= \{s \in l(u) \mid \delta(s, a) \in l(w)\} \\ &= l(u) \cap \delta^{-1}(l(w), a). \end{aligned} \tag{4.1}$$

In order to perform the same splits in each layer as before, we maintain a list L_k of k-candidates. We keep the list in order of the construction of nodes, because when we split w.r.t. a child of a node u before we split w.r.t. u, the result is not well-defined. Indeed, the order on L_k is the same as the order used by Algorithm 4.2. So far, the improved algorithm still would have time complexity $\mathcal{O}(mn)$.

To reduce the complexity we have to use Hopcroft's second idea of *processing the smaller half*. The key idea is that, when we fix a k-candidate v, all leaves are split with respect to (v, a) simultaneously. Instead of iterating over of all leaves to refine them, we iterate over $s \in \delta^{-1}(l(w), a)$ for all w in $C(v)$ and look up in which leaf it is contained to move s out of it. From Lemma 8 by Knuutila (2001) it follows that we can skip one of the children of v. This lowers the time complexity to $\mathcal{O}(m \log n)$. In order to move s out of its leaf, each leaf u is associated with a set of temporary children $C'(u)$ that is initially empty, and will be finalised after iterating over all s and w.

In Algorithm 4.4 we use the ideas described above. For each k-candidate v and input a, we consider all children w of v, except for the largest one (in case of multiple largest children, we skip one of these arbitrarily). For each state $s \in \delta^{-1}(l(w), a)$ we consider the leaf u containing it. If this leaf does not have an associated temporary

Require: A k-stable and minimal splitting tree T, and a list L_k
Ensure: T is a $(k+1)$-stable and minimal splitting tree, and a list L_{k+1}
1 $L_{k+1} \leftarrow \emptyset$
2 **for all** k-candidates v in L_k in order **do**
3 let w' be a node in $C(v)$ with $|l(w')| \geqslant |l(w)|$ for all nodes $w \in C(v)$
4 **for all** inputs a in I **do**
5 **for all** nodes w in $C(v) \setminus \{w'\}$ **do**
6 **for all** states s in $\delta^{-1}(l(w), a)$ **do**
7 locate leaf u such that $s \in l(u)$
8 **if** $C'(u)$ does not contain node u_w **then**
9 add a new node u_w to $C'(u)$
10 **end if**
11 move s from $l(u)$ to $l(u_w)$
12 **end for**
13 **end for**
14 **for all** leaves u with $C'(u) \neq \emptyset$**do**
15 **if** $|l(u)| = 0$ **then**
16 **if** $|C'(u)| = 1$ **then**
17 recover u by moving its elements back and clear $C'(u)$
18 **continue** with the next leaf
19 **end if**
20 set $p = u$ and $C(u) = C'(u)$
21 **else**
22 construct a new node p and set $C(p) = C'(u) \cup \{u\}$
23 insert p in the tree in the place where u was
24 **end if**
25 set $\sigma(p) = a\sigma(v)$
26 append p to L_{k+1} and clear $C'(u)$
27 **end for**
28 **end for**
29 **end for**

Algorithm 4.4 A better step towards the stability of a splitting tree.

child for w we create such a child (line 9), if this child exists we move s into that child (line 11).

Once we have done the simultaneous splitting for the candidate v and input a, we finalise the temporary children. This is done at lines 14–26. If there is only one temporary child with all the states, no split has been made and we recover this node (line 17). In the other case we make the temporary children permanent.

The states remaining in u are those for which $\delta(s, a)$ is in the child of v that we have skipped; therefore we will call it the *implicit child*. We should not touch these states to keep the theoretical time bound. Therefore, we construct a new parent node p that will "adopt" the children in $C'(u)$ together with u (line 20).

We will now explain why considering all but the largest children of a node lowers the algorithm's time complexity. Let T be a splitting tree in which we colour all children of each node blue, except for the largest one. Then:

Lemma 13. A state s is in at most $(\log_2 n) - 1$ labels of blue nodes.

Proof. Observe that every blue node u has a sibling u' such that $|l(u')| \geqslant |l(u)|$. So the parent $p(u)$ has at least $2|l(u)|$ states in its label, and the largest blue node has at most $n/2$ states.

Suppose a state s is contained in m blue nodes. When we walk up the tree starting at the leaf containing s, we will visit these m blue nodes. With each visit we can double the lower bound of the number of states. Hence $n/2 \geqslant 2^m$ and $m \leqslant (\log_2 n) - 1$. □

Corollary 14. A state s is in at most $\log_2 n$ sets $\delta^{-1}(l(u), a)$, where u is a blue node and a is an input in I.

If we now quantify over all transitions, we immediately get the following result. We note that the number of blue nodes is at most $n - 1$, but since this fact is not used, we leave this to the reader.

Corollary 15. Let $\mathcal{B}$ denote the set of blue nodes and define

$$\mathcal{X} = \{(b, a, s) \mid b \in \mathcal{B}, a \in I, s \in \delta^{-1}(l(b), a)\}.$$

Then $\mathcal{X}$ has at most $m \log_2 n$ elements.

The important observation is that when using Algorithm 4.4 we iterate in total over every element in $\mathcal{X}$ at most once.

Theorem 16. Algorithm 4.2 using Algorithm 4.4 runs in $\mathcal{O}(m \log n)$ time.

Proof. We prove that bookkeeping does not increase time complexity by discussing the implementation.

Inverse transition. δ^{-1} can be constructed as a preprocessing step in $\mathcal{O}(m)$.

State sorting. As described in Section 1.2, we maintain a refinable partition data structure. Each time new pair of a k-candidate v and input a is considered, leaves are split by performing a bucket sort.

First, buckets are created for each node in $w \in C(v) \setminus w'$ and each leaf u that contains one or more elements from $\delta^{-1}(l(w), a)$, where w' is a largest child of v. The buckets are filled by iterating over the states in $\delta^{-1}(l(w), a)$ for all w. Then, a pivot is

set for each leaf u such that exactly the states that have been placed in a bucket can be moved right of the pivot (and untouched states in $\delta^{-1}(l(w'), a)$ end up left of the pivot). For each leaf u, we iterate over the states in its buckets and the corresponding indices right of its pivot, and we swap the current state with the one that is at the current index. For each bucket a new leaf node is created. The refinable partition is updated such that the current state points to the most recently created leaf.

This way, we assure constant time lookup of the leaf for a state, and we can update the array in constant time when we move elements out of a leaf.

Largest child. For finding the largest child, we maintain counts for the temporary children and a current biggest one. On finalising the temporary children we store (a reference to) the biggest child in the node, so that we can skip this node later in the algorithm.

Storing sequences. The operation on line 25 is done in constant time by using a linked list. □

4 Application in Conformance Testing

A splitting tree can be used to extract relevant information for two classical test generation methods: a *characterisation set* for the W-method and a *separating family* for the HSI-method. For an introduction and comparison of FSM-based test generation methods we refer to Dorofeeva, et al. (2010) or Chapter 2.

Definition 17. A set $W \subset I^*$ is called a *characterisation set* if for every pair of inequivalent states s, t there is a sequence $w \in W$ such that $\lambda(s, w) \neq \lambda(t, w)$.

Lemma 18. Let T be a complete splitting tree, then the set $\{\sigma(u) \mid u \in T\}$ is a characterisation set.

Proof. Let $W = \{\sigma(u) \mid u \in T\}$. Let $s, t \in S$ be inequivalent states, then by completeness s and t are contained in different leaves of T. Hence $u = lca(s, t)$ exists and $\sigma(u) \in W$ separates s and t. This shows that W is a characterisation set. □

Lemma 19. A characterisation set with minimal length sequences can be constructed in time $\mathcal{O}(m \log n)$.

Proof. By Lemma 18 the sequences associated with the inner nodes of a splitting tree form a characterisation set. By Theorem 16, such a tree can be constructed in time $\mathcal{O}(m \log n)$. Traversing the tree to obtain the characterisation set is linear in the number of nodes (and hence linear in the number of states). □

Definition 20. A collection of sets $\{H_s\}_{s\in S}$ is called a *separating family* if for every pair of inequivalent states s, t there is a sequence h such that $\lambda(s,h) \neq \lambda(t,h)$ and h is a prefix of some $h_s \in H_s$ and some $h_t \in H_t$.

Lemma 21. Let T be a complete splitting tree, the sets $\{\sigma(u) \mid s \in l(u), u \in T\}_{s\in S}$ form a separating family.

Proof. Let $H_s = \{\sigma(u) \mid s \in l(u)\}$. Let $s, t \in S$ be inequivalent states, then by completeness s and t are contained in different leaves of T. Hence $u = lca(s,t)$ exists. Since both s and t are contained in $l(u)$, the separating sequence $\sigma(u)$ is contained in both sets H_s and H_t. Therefore, it is a (trivial) prefix of some word $h_s \in H_s$ and some $h_t \in H_t$. Hence $\{H_s\}_{s\in S}$ is a separating family. □

Lemma 22. A separating family with minimal length sequences can be constructed in time $\mathcal{O}(m \log n + n^2)$.

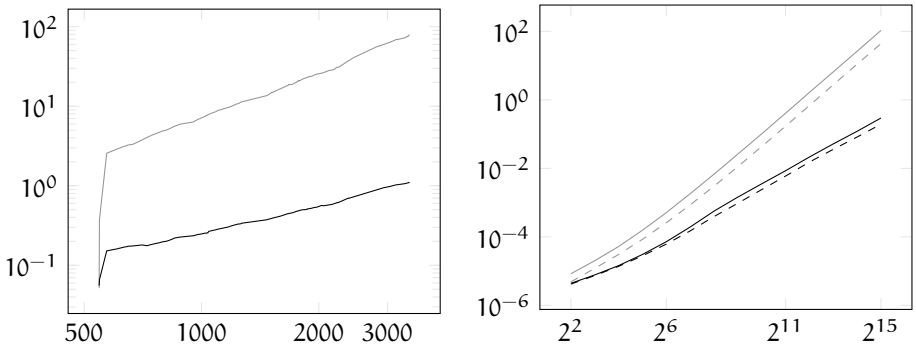
Proof. The separating family can be constructed from the splitting tree by collecting all sequences of all parents of a state (by Lemma 21). Since we have to do this for every state, this takes $\mathcal{O}(n^2)$ time. □

For test generation one also needs a transition cover. This can be constructed in linear time with a breadth first search. We conclude that we can construct all necessary information for the W-method in time $\mathcal{O}(m \log n)$ as opposed the the $\mathcal{O}(mn)$ algorithm used by Dorofeeva, et al. (2010). Furthermore, we conclude that we can construct all the necessary information for the HSI-method in time $\mathcal{O}(m \log n + n^2)$, improving on the the reported bound $\mathcal{O}(mn^3)$ by Hierons and Türker (2015). The original HSI-method was formulated differently and might generate smaller sets. We conjecture that our separating family has the same size if we furthermore remove redundant prefixes. This can be done in $\mathcal{O}(n^2)$ time using a trie data structure.

5 *Experimental Results*

We have implemented Algorithms 4.3 in Go, and we have compared their running time on two sets of FSMs.[16] The first set is from Smeenk, et al. (2015a), where FSMs for embedded control software were automatically constructed. These FSMs are of increasing size, varying from 546 to 3 410 states, with 78 inputs and up to 151 outputs. The second set is inferred from Hopcroft (1971), where two classes of finite automata, A and B, are described that serve as a worst case for Algorithms 4.3 respectively. The FSMs that we have constructed for these automata have 1 input, 2 outputs, and $2^2 - 2^{15}$ states. The running times in seconds on an Intel Core i5-2500 are plotted in Figure 4.3. We note that different slopes imply different complexity classes, since both axes have a logarithmic scale.

[16] Available at https://github.com/Jaxan/partition.



(a) Embedded control software.

(b) Class A (dashed) and class B (solid).

Figure 4.3 Running time in seconds of Algorithm 4.3 in grey and Algorithm 4.4 in black.

6 Conclusion

In this chapter we have described an efficient algorithm for constructing a set of minimal-length sequences that pairwise distinguish all states of a finite state machine. By extending Hopcroft's minimisation algorithm, we are able to construct such sequences in $\mathcal{O}(m \log n)$ for a machine with m transitions and n states. This improves on the traditional $\mathcal{O}(mn)$ method that is based on the classic algorithm by Moore. As an upshot, the sequences obtained form a characterisation set and a separating family, which play a crucial in conformance testing.

Two key observations were required for a correct adaptation of Hopcroft's algorithm. First, it is required to perform splits in order of the length of their associated sequences. This guarantees minimality of the obtained separating sequences. Second, it is required to consider nodes as a candidate before any one of its children are considered as a candidate. This order follows naturally from the construction of a splitting tree.

Experimental results show that our algorithm outperforms the classic approach for both worst-case finite state machines and models of embedded control software. Applications of minimal separating sequences such as the ones described by Dorofeeva, et al. (2010) and Smeenk, et al. (2015a) therefore show that our algorithm is useful in practice.

Part 2:
Nominal Techniques

Chapter 5
Learning Nominal Automata

Joshua Moerman
Radboud University

Matteo Sammartino
University College London

Alexandra Silva
University College London

Bartek Klin
University of Warsaw

Michał Szynwelski
University of Warsaw

Abstract

We present an Angluin-style algorithm to learn nominal automata, which are acceptors of languages over infinite (structured) alphabets. The abstract approach we take allows us to seamlessly extend known variations of the algorithm to this new setting. In particular, we can learn a subclass of nominal non-deterministic automata. An implementation using a recently developed Haskell library for nominal computation is provided for preliminary experiments.

This chapter is based on the following publication:
Moerman, J., Sammartino, M., Silva, A., Klin, B., & Szynwelski, M. (2017). Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*. ACM. doi:10.1145/3009837.3009879

Automata are a well established computational abstraction with a wide range of applications, including modelling and verification of (security) protocols, hardware, and software systems. In an ideal world, a model would be available before a system or protocol is deployed in order to provide ample opportunity for checking important properties that must hold and only then the actual system would be synthesised from the verified model. Unfortunately, this is not at all the reality: Systems and protocols are developed and coded in short spans of time and if mistakes occur they are most likely found after deployment. In this context, it has become popular to infer or learn a model from a given system just by observing its behaviour or response to certain queries. The learned model can then be used to ensure the system is complying to desired properties or to detect bugs and design possible fixes.

Automata learning, or regular inference, is a widely used technique for creating an automaton model from observations. The original algorithm by Angluin (1987) works for deterministic finite automata, but since then has been extended to other types of automata, including Mealy machines and I/O automata (see Niese, 2003, §8.5, and Aarts & Vaandrager, 2010), and even a special class of context-free grammars (see Isberner, 2015, §6) Angluin's algorithm is sometimes referred to as *active learning*, because it is based on direct interaction of the learner with an oracle ("the Teacher") that can answer different types of queries. This is in contrast with *passive* learning, where a fixed set of positive and negative examples is given and no interaction with the system is possible.

In this chapter, staying in the realm of active learning, we will extend Angluin's algorithm to a richer class of automata. We are motivated by situations in which a program model, besides control flow, needs to represent basic data flow, where data items are compared for equality (or for other theories such as total ordering). In these situations, values for individual symbols are typically drawn from an infinite domain and automata over *infinite alphabets* become natural models, as witnessed by a recent trend (Aarts, et al., 2015; Bojańczyk, et al., 2014; Bollig, et al., 2013; Cassel, et al., 2016; D'Antoni & Veanes, 2014).

One of the foundational approaches to formal language theory for infinite alphabets uses the notion of nominal sets (Bojańczyk, et al., 2014). The theory of nominal sets originates from the work of Fraenkel in 1922, and they were originally used to prove the independence of the axiom of choice and other axioms. They have been rediscovered in Computer Science by Gabbay and Pitts (see Pitts, 2013 for historical notes), as an elegant formalism for modelling name binding, and since then they form the basis of many research projects in the semantics and concurrency community. In a nutshell, nominal sets are infinite sets equipped with symmetries which make them finitely representable and tractable for algorithms. We make crucial use of this feature in the development of a learning algorithm.

Our main contributions are the following.

– A generalisation of Angluin's original algorithm to nominal automata. The generalisation follows a generic pattern for transporting computation models from

finite sets to nominal sets, which leads to simple correctness proofs and opens the door to further generalisations. The use of nominal sets with different symmetries also creates potential for generalisation, e.g., to languages with time features (Bojańczyk & Lasota, 2012) or data dependencies represented as graphs (Montanari & Sammartino, 2014).
– An extension of the algorithm to nominal non-deterministic automata (nominal NFAs). To the best of our knowledge, this is the first learning algorithm for non-deterministic automata over infinite alphabets. It is important to note that, in the nominal setting, NFAs are strictly more expressive than DFAs. We learn a subclass of the languages accepted by nominal NFAs, which includes all the languages accepted by nominal DFAs. The main advantage of learning NFAs directly is that they can provide exponentially smaller automata when compared to their deterministic counterpart. This can be seen both as a generalisation and as an optimisation of the algorithm.
– An implementation using a recently developed Haskell library tailored to nominal computation – NLambda, or Nλ, by Klin and Szynwelski (2016). Our implementation is the first non-trivial application of a novel programming paradigm of functional programming over infinite structures, which allows the programmer to rely on convenient intuitions of searching through infinite sets in finite time.

This chapter is organised as follows. In Section 1, we present an overview of our contributions (and the original algorithm) highlighting the challenges we faced in the various steps. In Section 2, we revise some basic concepts of nominal sets and automata. Section 3 contains the core technical contributions: The new algorithm and proof of correctness. In Section 4, we describe an algorithm to learn nominal non-deterministic automata. Section 5 contains a description of NLambda, details of the implementation, and results of preliminary experiments. Section 6 contains a discussion of related work. We conclude this chapter with a discussion section where also future directions are presented.

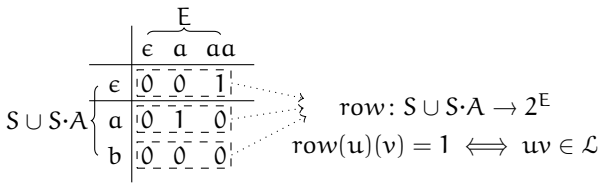
1 *Overview of the Approach*

In this section, we give an overview through examples. We will start by explaining the original algorithm for regular languages over finite alphabets, and then explain the challenges in extending it to nominal languages.

Angluin's algorithm L^* provides a procedure to learn the minimal DFA accepting a certain (unknown) language $\mathcal{L}$. The algorithm has access to a *teacher* which answers two types of queries:
– *membership queries*, consisting of a single word $w \in A^*$, to which the teacher will reply whether $w \in \mathcal{L}$ or not;

– *equivalence queries*, consisting of a hypothesis DFA H, to which the teacher replies **yes** if $\mathcal{L}(H) = \mathcal{L}$, and **no** otherwise, providing a counterexample $w \in \mathcal{L}(H) \triangle \mathcal{L}$ (where $\triangle$ denotes the symmetric difference of two languages).

The learning algorithm works by incrementally building an *observation table*, which at each stage contains partial information about the language $\mathcal{L}$. The algorithm is able to fill the table with membership queries. As an example, and to set notation, consider the following table (over the alphabet $A = \{a, b\}$).



This table indicates that $\mathcal{L}$ contains at least aa and definitely does not contain the words $\epsilon, a, b, ba, baa, aaa$. Since row is fully determined by the language $\mathcal{L}$, we will from now on refer to an observation table as a pair (S, E), leaving the language $\mathcal{L}$ implicit.

Given an observation table (S, E) one can construct a deterministic automaton $M(S, E) = (Q, q_0, \delta, F)$ where

– $Q = \{row(s) \mid s \in S\}$ is a finite set of states;
– $F = \{row(s) \mid s \in S, row(s)(\epsilon) = 1\} \subseteq Q$ is the set of final states;
– $q_0 = row(\epsilon)$ is the initial state;
– $\delta \colon Q \times A \to Q$ is the transition function given by $\delta(row(s), a) = row(sa)$.

For this to be well-defined, we need to have $\epsilon \in S$ (for the initial state) and $\epsilon \in E$ (for final states), and for the transition function there are two crucial properties of the table that need to hold: Closedness and consistency. An observation table (S, E) is *closed* if for all $t \in S \cdot A$ there exists an $s \in S$ such that $row(t) = row(s)$. An observation table (S, E) is *consistent* if, whenever s_1 and s_2 are elements of S such that $row(s_1) = row(s_2)$, for all $a \in A$, $row(s_1 a) = row(s_2 a)$. Each time the algorithm constructs an automaton, it poses an equivalence query to the teacher. It terminates when the answer is **yes**, otherwise it extends the table with the counterexample provided.

1.1 Simple Example of Execution

Angluin's algorithm is displayed in Algorithm 5.1. Throughout this section, we will consider the language(s)

$$\mathcal{L}_n = \{ww \mid w \in A^*, |w| = n\}.$$

If the alphabet A is finite then $\mathcal{L}_n$ is regular for any $n \in \mathbb{N}$, and there is a finite DFA accepting it.

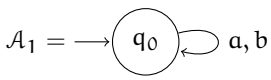
```
1   S, E ← {ε}
2   repeat
3     while (S, E) is not closed or not consistent do
4       if (S, E) is not closed then
5         find s1 ∈ S, a ∈ A such that row(s1 a) ≠ row(s) for all s ∈ S
6         S ← S ∪ {s1 a}
7       end if
8       if (S, E) is not consistent then
9         find s1, s2 ∈ S, a ∈ A, and e ∈ E such that
10                                row(s1) = row(s2) and L(s1 a e) ≠ L(s2 a e)
11        E ← E ∪ {a e}
12      end if
13    end while
14    Make the conjecture M(S, E)
15    if the Teacher replies no, with a counter-example t then
16      S ← S ∪ pref(t)
17    end if
18  until the Teacher replies yes to the conjecture M(S, E)
19  return M(S, E)
```

Algorithm 5.1 The L* learning algorithm from Angluin (1987).

The language $\mathcal{L}_1 = \{aa, bb\}$ looks trivial, but the minimal DFA recognising it has as many as 5 states. Angluin's algorithm will terminate in (at most) 5 steps. We illustrate some relevant ones.

Step 1 We start from $S, E = \{\epsilon\}$, and we fill the entries of the table below by asking membership queries for ϵ, a and b. The table is closed and consistent, so we construct the hypothesis $\mathcal{A}_1$, where $q_0 = row(\epsilon) = \{\epsilon \mapsto 0\}$:

	ϵ
ϵ	0
a	0
b	0



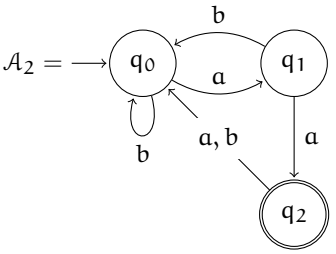
The Teacher replies **no** and gives the counterexample aa, which is in $\mathcal{L}_1$ but it is not accepted by $\mathcal{A}_1$. Therefore, line 16 of the algorithm is triggered and we set $S = \{\epsilon, a, aa\}$.

Step 2 The table becomes the one on the left below. It is closed, but not consistent: Rows ϵ and a are identical, but appending a leads to different rows, as depicted.

Therefore, line 10 is triggered and an extra column a, highlighted in red, is added. The new table is closed and consistent and a new hypothesis $\mathcal{A}_2$ is constructed.

	ϵ
ϵ	0
a	0
aa	1
b	0
ab	0
aaa	0
aab	0

	ϵ	a
ϵ	0	0
a	0	1
aa	1	0
b	0	0
ab	0	0
aaa	0	0
aab	0	0

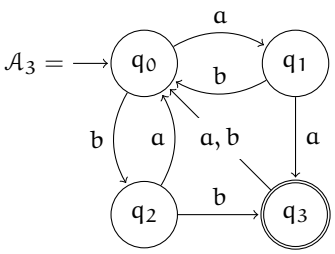


The Teacher again replies **no** and gives the counterexample bb, which should be accepted by $\mathcal{A}_2$ but it is not. Therefore we put $S \leftarrow S \cup \{b, bb\}$.

Step 3 The new table is the one on the left. It is closed, but ϵ and b violate consistency, when b is appended. Therefore we add the column b and we get the table on the right, which is closed and consistent. The new hypothesis is $\mathcal{A}_3$.

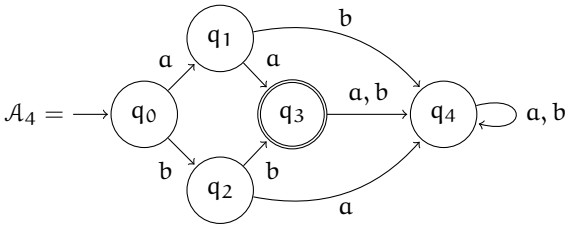
	ϵ	a
ϵ	0	0
a	0	1
aa	1	0
b	0	0
bb	1	0
ab	0	0
aaa	0	0
aab	0	0
ba	0	0
bba	0	0
bbb	0	0

	ϵ	a	b
ϵ	0	0	0
a	0	1	0
aa	1	0	0
b	0	0	1
bb	1	0	0
ab	0	0	0
aaa	0	0	0
aab	0	0	0
ba	0	0	0
bba	0	0	0
bbb	0	0	0



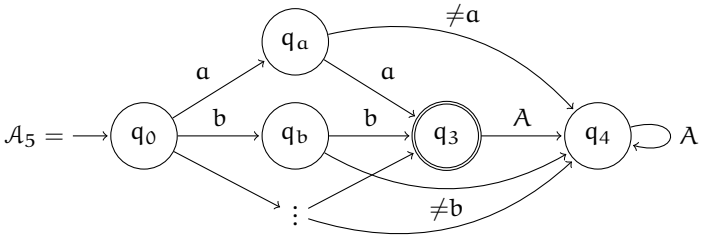
The Teacher replies **no** and provides the counterexample babb, so $S \leftarrow S \cup \{ba, bab\}$.

Step 4 One more step brings us to the correct hypothesis $\mathcal{A}_4$ (details are omitted).

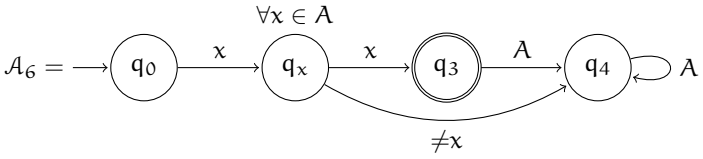


1.2 *Learning Nominal Languages*

Consider now an infinite alphabet $A = \{a, b, c, d, ...\}$. The language $\mathcal{L}_1$ becomes $\{aa, bb, cc, dd, ...\}$. Classical theory of finite automata does not apply to this kind of languages, but one may draw an infinite deterministic automaton that recognises $\mathcal{L}_1$ in the standard sense:



where $\xrightarrow{A}$ and $\xrightarrow{\neq a}$ stand for the infinitely-many transitions labelled by elements of A and $A \setminus \{a\}$, respectively. This automaton is infinite, but it can be finitely presented in a variety of ways, for example:



One can formalise the quantifier notation above (or indeed the "dots" notation above that) in several ways. A popular solution is to consider finite *register automata* (Demri & Lazic, 2009 and Kaminski & Francez, 1994), i.e., finite automata equipped with a finite number of registers where alphabet letters can be stored and later compared for equality. Our language $\mathcal{L}_1$ is recognised by a simple automaton with four states and one register. The problem of learning registered automata has been successfully attacked before by, for instance, Howar, et al. (2012).

In this chapter, however, we consider nominal automata by Bojańczyk, et al. (2014) instead. These automata ostensibly have infinitely many states, but the set of states can be finitely presented in a way open to effective manipulation. More specifically, in a nominal automaton the set of states is subject to an action of permutations of a set $\mathbb{A}$ of *atoms*, and it is finite up to that action. For example, the set of states of $\mathcal{A}_5$ is:

$$\{q_0, q_3, q_4\} \cup \{q_a \mid a \in A\}$$

and it is equipped with a canonical action of permutations $\pi \colon \mathbb{A} \to \mathbb{A}$ that maps every q_a to $q_{\pi(a)}$ and leaves q_0, q_3 and q_4 fixed. Technically speaking, the set of states has four *orbits* (one infinite orbit and three fixed points) of the action of the group of permutations of $\mathbb{A}$. Moreover, it is required that in a nominal automaton the transition

relation is *equivariant*, i.e., closed under the action of permutations. The automaton $\mathcal{A}_5$ has this property: For example, it has a transition $q_a \xrightarrow{a} q_3$, and for any $\pi\colon \mathbb{A} \to \mathbb{A}$ there is also a transition $\pi(q_a) = q_{\pi(a)} \xrightarrow{\pi(a)} q_3 = \pi(q_3)$.

Nominal automata with finitely many orbits of states are equi-expressive with finite register automata (Bojańczyk, et al., 2014), but they have an important theoretical advantage: They are a direct reformulation of the classical notion of finite automaton, where one replaces finite sets with orbit-finite sets and functions (or relations) with equivariant ones. A research programme advocated by Bojańczyk, et al. is to transport various computation models, algorithms and theorems along this correspondence. This can often be done with remarkable accuracy, and our results are a witness to this. Indeed, as we shall see, nominal automata can be learned with an algorithm that is almost a verbatim copy of the classical Angluin's one.

Indeed, consider applying Angluin's algorithm to our new language $\mathcal{L}_1$. The key idea is to change the basic data structure: Our observation table (S, E) will be such that S *and* E *are equivariant subsets of* A^*, i.e., they are closed under the canonical action of atom permutations. In general, such a table has *infinitely many rows and columns*, so the following aspects of Algorithm 5.1 seem problematic:

– line 4 and line 8: finding witnesses for closedness or consistency violations potentially require checking all infinitely many rows;
– line 16: every counterexample t has infinitely many prefixes, so it is not clear how one constructs an infinite set S in finite time. However, an infinite S is necessary for the algorithm to ever succeed, because no finite automaton recognises $\mathcal{L}_1$.

At this stage, we need to observe that due to equivariance of S, E and $\mathcal{L}_1$, the following crucial properties hold:

(P1) the sets S, $S{\cdot}A$ and E admit a *finite* representation up to permutations;

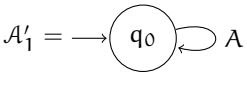
(P2) the function row is such that $row(\pi(s))(\pi(e)) = row(s)(e)$, for all $s \in S$ and $e \in E$, so the observation table admits a finite symbolic representation.

Intuitively, checking closedness and consistency, and finding a witness for their violations, can be done effectively on the representations up to permutations **(P1)**. This is sound, as row is invariant w.r.t. permutations **(P2)**.

We now illustrate these points through a few steps of the algorithm for $\mathcal{L}_1$.

Step 1′ We start from $S, E = \{\epsilon\}$. We have $S{\cdot}A = A$, which is infinite but admits a finite representation. In fact, for any $a \in A$, we have $A = \{\pi(a) \mid \pi \text{ is a permutation}\}$. Then, by **(P2)**, $row(\pi(a))(\epsilon) = row(a)(\epsilon) = 0$, for all π, so the first table can be written as:

	ϵ
ϵ	0
a	0



It is closed and consistent. Our hypothesis is $\mathcal{A}'_1$, where $\delta_{\mathcal{A}'_1}(row(\epsilon), x) = row(x) = q_0$, for all $x \in A$. As in **Step 1**, the Teacher replies with the counterexample aa.

Step 2' By equivariance of $\mathcal{L}_1$, the counterexample tells us that *all* words of length 2 with two repeated letters are accepted. Therefore we extend S with the (infinite!) set of such words. The new symbolic table is depcited on the right.

	ϵ
ϵ	0
a	0
aa	1
ab	0
aaa	0
aab	0

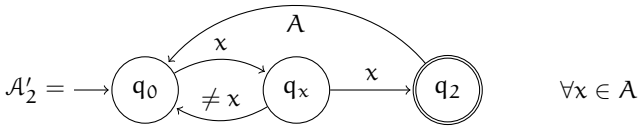
The lower part stands for elements of $S \cdot A$. For instance, ab stands for words obtained by appending a fresh letter to words of length 1 (row a). It can be easily verified that all cases are covered. Notice that the table is different from that of **Step 2**: A single b is not in the lower part, because it can be obtained from a via a permutation. The table is closed.

Now, for consistency we need to check $row(\epsilon x) = row(ax)$, for all $a, x \in A$. Again, by **(P2)**, it is enough to consider rows of the table above. Consistency is violated, because $row(a) \neq row(aa)$. We found a "symbolic" witness a for such violation. In order to fix consistency, while keeping E equivariant, we need to add columns for all $\pi(a)$. The resulting table is

	ϵ	a	b	c	...
ϵ	0	0	0	0	...
a	0	1	0	0	...
aa	1	0	0	0	...
ab	0	0	0	0	...
aaa	0	0	0	0	...
aab	0	0	0	0	...

where non-specified entries are 0. Only finitely many entries of the table are relevant: $row(s)$ is fully determined by its values on letters in s and on just one letter not in s. For instance, we have $row(a)(a) = 1$ and $row(a)(a') = 0$, for all $a' \in A \setminus \{a\}$. The table is trivially consistent.

Notice that this step encompasses both **Step 2** and **3**, because the rows b and bb added by **Step 2** are already represented by a and aa. The hypothesis automaton is



This is again incorrect, but one additional step will give the correct hypothesis automaton $\mathcal{A}_6$.

1.3 Generalisation to Non-Deterministic Automata

Since our extension of Angluin's $\mathtt{L}^*$ algorithm stays close to her original development, exploring extensions of other variations of $\mathtt{L}^*$ to the nominal setting can be done in a systematic way. We will show how to extend the algorithm $\mathtt{NL}^*$ for learning NFAs by Bollig, et al. (2009). This has practical implications: It is well-known that NFAs are exponentially more succinct than DFAs. This is true also in the nominal setting. However, there are challenges in the extension that require particular care.

– Nominal NFAs are strictly more expressive than nominal DFAs. We will show that the nominal version of $\mathtt{NL}^*$ terminates for all nominal NFAs that have a corresponding nominal DFA and, more surprisingly, that it is capable of learning some languages that are not accepted by nominal DFAs.
– Language equivalence of nominal NFAs is undecidable. This does not affect the correctness proof, as it assumes a teacher which is able to answer equivalence queries accurately. For our implementation, we will describe heuristics that produce correct results in many cases.

For the learning algorithm the power of non-determinism means that we can make some shortcuts during learning: If we want to make the table closed, we were previously required to find an equivalent row in the upper part; now we may find a sum of rows which, together, are equivalent to an existing row. This means that in some cases fewer rows will be added for closedness.

2 Preliminaries

We recall the notions of nominal sets, nominal automata and nominal regular languages. We refer to Bojańczyk, et al. (2014) for a detailed account.

Let $\mathbb{A}$ be a countable set and let $\mathtt{Perm}(\mathbb{A})$ be the set of *permutations on* $\mathbb{A}$, i.e., the bijective functions $\pi\colon \mathbb{A} \to \mathbb{A}$. Permutations form a group where the identity permutation $\mathtt{id}$ is the unit element, inverse is functional inverse and multiplication is function composition.

A *nominal set* (Pitts, 2013) is a set X together with a function $\cdot\colon \mathtt{Perm}(\mathbb{A}) \times X \to X$, interpreting permutations over X. Such function must be a *group action* of $\mathtt{Perm}(\mathbb{A})$, i.e., it must satisfy $\mathtt{id} \cdot x = x$ and $\pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x$. We say that a finite $A \subset \mathbb{A}$ *supports* $x \in X$ whenever, for all π acting as the identity on A, we have $\pi \cdot x = x$. In other words, permutations that only move elements outside A do not affect x. The support of $x \in X$, denoted $\mathtt{supp}(x)$, is the smallest finite set supporting x. We require nominal sets to have *finite support*, meaning that $\mathtt{supp}(x)$ exists for all $x \in X$.

The *orbit of* x, denoted $\mathtt{orb}(x)$, is the set of elements in X reachable from x via permutations, explicitly

$$\mathtt{orb}(x) = \{\pi \cdot x \mid \pi \in \mathtt{Perm}(\mathbb{A})\}.$$

We say that X is *orbit-finite* whenever it is a union of finitely many orbits.

Given a nominal set X, a subset $Y \subseteq X$ is *equivariant* if it is preserved by permutations, i.e., $\pi \cdot y \in Y$, for all $y \in Y$. In other words, Y is a union of some orbits of X. This definition extends to the notion of an equivariant relation $R \subseteq X \times Y$, by setting $\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$, for $(x, y) \in R$; similarly for relations of greater arity. The *dimension* of nominal set X is the maximal size of $\mathrm{supp}(x)$, for any $x \in X$. Every orbit-finite set has finite dimension.

We define $\mathbb{A}^{(k)} = \{(a_1, ..., a_k) \mid a_i \neq a_j \text{ for } i \neq j\}$. For every single-orbit nominal set X with dimension k, there is a surjective equivariant map

$$f_X \colon \mathbb{A}^{(k)} \to X.$$

This map can be used to get an upper bound for the number of orbits of $X_1 \times ... \times X_n$, for X_i a nominal set with l_i orbits and dimension k_i. Suppose O_i is an orbit of X_i. Then we have a surjection

$$f_{O_1} \times \cdots \times f_{O_n} \colon \mathbb{A}^{(k_i)} \times \cdots \times \mathbb{A}^{(k_n)} \to O_1 \times \cdots \times O_n$$

stipulating that the codomain cannot have more orbits than the domain. Let $f_{\mathbb{A}}(\{k_i\})$ denote the number of orbits of $\mathbb{A}^{(k_1)} \times \cdots \times \mathbb{A}^{(k_n)}$, for any finite sequence of natural numbers $\{k_i\}$. We can form at most $l = l_1 l_2 ... l_n$ tuples of the form $O_1 \times \cdots \times O_n$, so $X_1 \times \cdots \times X_n$ has at most $l f_{\mathbb{A}}(k_1, ..., k_n)$ orbits.

For X single-orbit, the *local symmetries* are defined by the group

$$\{g \in S_k \mid f_X(x_1, ..., x_k) = f_X(x_{g(1)}, ..., x_{g(k)}) \text{ for all } x_i \in X\},$$

where k is the dimension of X and S_k is the *symmetric group* of permutations over k distinct elements.

NFAs on sets have a finite state space. We can define *nominal* NFAs, with the requirement that the state space is orbit-finite and the transition relation is equivariant. A nominal NFA is a tuple (Q, A, Q_0, F, δ), where:

- Q is an orbit-finite nominal set of *states*;
- A is an orbit-finite nominal alphabet;
- $Q_0, F \subseteq Q$ are equivariant subsets of *initial* and *final states*;
- $\delta \subseteq Q \times A \times Q$ is an equivariant *transition relation*.

A nominal DFA is a special case of nominal NFA where $Q_0 = \{q_0\}$ and the transition relation is an equivariant function $\delta \colon Q \times A \to Q$. Equivariance here can be rephrased as requiring $\delta(\pi \cdot q, \pi \cdot a) = \pi \cdot \delta(q, a)$. In most examples we take the alphabet to be $A = \mathbb{A}$, but it can be any orbit-finite nominal set. For instance, $A = \mathrm{Act} \times \mathbb{A}$, where Act is a finite set of actions, represents actions $\mathrm{act}(x)$ with one parameter $x \in \mathbb{A}$ (actions with arity n can be represented via n-fold products of $\mathbb{A}$).

A language $\mathcal{L}$ is *nominal regular* if it is recognised by a nominal DFA. The theory of nominal regular languages recasts the classical one using nominal concepts. A nominal Myhill-Nerode-style *syntactic congruence* is defined: $w, w' \in A^*$ are *equivalent* w.r.t. $\mathcal{L}$, written $w \equiv_{\mathcal{L}} w'$, whenever

$$wv \in \mathcal{L} \iff w'v \in \mathcal{L}$$

for all $v \in A^*$. This relation is equivariant and the set of equivalence classes $[w]_{\mathcal{L}}$ is a nominal set.

Theorem 1. (Myhill-Nerode theorem for nominal sets by Bojańczyk, et al., 2014) Let $\mathcal{L}$ be a nominal language. The following conditions are equivalent:
1. the set of equivalence classes of $\equiv_{\mathcal{L}}$ is orbit-finite;
2. $\mathcal{L}$ is recognised by a nominal DFA.

Unlike what happens for ordinary regular languages, nominal NFAs and nominal DFAs *are not equi-expressive*. Here is an example of a language accepted by a nominal NFA, but not by a nominal DFA:

$$\mathcal{L}_{eq} = \{a_1 ... a_n \mid a_i = a_j, \text{for some } i < j \in \{1, ..., n\}\}.$$

In the theory of nominal regular languages, several problems are decidable: Language inclusion and minimality test for nominal DFAs. Moreover, orbit-finite nominal sets can be finitely-represented, and so can be manipulated by algorithms. This is the key idea underpinning our implementation.

2.1 *Different Atom Symmetries*

An important advantage of nominal set theory as considered by Bojańczyk, et al. (2014) is that it retains most of its properties when the structure of atoms $\mathbb{A}$ is replaced with an arbitrary infinite relational structure subject to a few model-theoretic assumptions. An example alternative structure of atoms is the total order of rational numbers $(\mathbb{Q}, <)$, with the group of monotone bijections of $\mathbb{Q}$ taking the role of the group of all permutations. The theory of nominal automata remains similar, and an example nominal language over the atoms $(\mathbb{Q}, <)$ is:

$$\{a_1 ... a_n \mid a_i \leqslant a_j, \text{for some } i < j \in \{1, ..., n\}\}$$

which is recognised by a nominal DFA over those atoms.

To simplify the presentation, in this chapter we concentrate on the "equality atoms" only. However, both the theory and the implementation can be generalised to other atom structures, with the "ordered atoms" $(\mathbb{Q}, <)$ as the simplest other example. We investigate the total order symmetry $(\mathbb{Q}, <)$ in Chapter 6.

3 *Angluin's Algorithm for Nominal DFAs*

In our algorithm, we will assume a teacher as described at the start of Section 1. In particular, the teacher is able to answer membership queries and equivalence queries,

now in the setting of nominal languages. We fix a target language $\mathcal{L}$, which is assumed to be a nominal regular language.

The learning algorithm for nominal automata, νL^*, will be very similar to L^* in Algorithm 5.1. In fact, we only change the following lines:

$$\begin{array}{ll} 6' & S \leftarrow S \cup \mathrm{orb}(sa) \\ 11' & E \leftarrow E \cup \mathrm{orb}(ae) \\ 16' & S \leftarrow S \cup \mathrm{pref}(\mathrm{orb}(t)) \end{array} \tag{5.1}$$

The basic data structure is an *observation table* (S, E, T) where S and E are orbit-finite subsets of A^* and $T\colon S \cup S{\cdot}A \times E \to 2$ is an equivariant function defined by $T(s, e) = \mathcal{L}(se)$ for each $s \in S \cup S{\cdot}A$ and $e \in E$. Since T is determined by $\mathcal{L}$ we omit it from the notation. Let $\mathrm{row}\colon S \cup S{\cdot}A \to 2^E$ denote the curried counterpart of T. Let $u \sim v$ denote the relation $\mathrm{row}(u) = \mathrm{row}(v)$.

Definition 2. The table is called *closed* if for each $t \in S{\cdot}A$ there is a $s \in S$ with $t \sim s$. The table is called *consistent* if for each pair $s_1, s_2 \in S$ with $s_1 \sim s_2$ we have $s_1 a \sim s_2 a$ for all $a \in A$.

The above definitions agree with the abstract definitions given by Jacobs and Silva (2014) and we may use some of their results implicitly. The intuition behind the definitions is as follows. Closedness assures us that for each state we have a successor state for each input. Consistency assures us that each state has at most one successor for each input. Together it allows us to construct a well-defined minimal automaton from the observations in the table.

The algorithm starts with a trivial observation table and tries to make it closed and consistent by adding orbits of rows and columns, filling the table via membership queries. When the table is closed and consistent it constructs a hypothesis automaton and poses an equivalence query.

The pseudocode for the nominal version is the same as listed in Algorithm 5.1, modulo the changes displayed in (5.1). However, we have to take care to ensure that all manipulations and tests on the (possibly) infinite sets S, E and A terminate in finite time. We refer to Bojańczyk, et al. (2014) and Pitts (2013) for the full details on how to represent these structures and provide a brief sketch here. The sets S, E, A and $S{\cdot}A$ can be represented by choosing a representative for each orbit. The function T in turn can be represented by cells $T_{i,j} : \mathrm{orb}(s_i) \times \mathrm{orb}(e_j) \to 2$ for each representative s_i and e_j. Note, however, that the product of two orbits may consist of several orbits, so that $T_{i,j}$ is not a single boolean value. Each cell is still orbit-finite and can be filled with only finitely many membership queries. Similarly the curried function row can be represented by a finite structure.

To check whether the table is closed, we observe that if we have a corresponding row $s \in S$ for some $t \in S{\cdot}A$, this holds for any permutation of t. Hence it is enough to check the following: For all representatives $t \in S{\cdot}A$ there is a representative $s \in S$

with $row(t) = \pi \cdot row(s)$ for some permutation π. Note that we only have to consider finitely many permutations, since the support is finite and so we can decide this property. Furthermore, if the property does not hold, we immediately find a witness represented by t.

Consistency is a bit more complicated, but it is enough to consider the set of inconsistencies, $\{(s_1, s_2, a, e) \mid row(s_1) = row(s_2) \wedge row(s_1 a)(e) \neq row(s_2 a)(e)\}$. It is an equivariant subset of $S \times S \times A \times E$ and so it is orbit-finite. Hence we can decide emptiness and obtain representatives if it is non-empty.

Constructing the hypothesis happens in the same way as before (Section 1), where we note the state space is orbit-finite since it is a quotient of S. Moreover, the function row is equivariant, so all structure (Q_0, F and δ) is equivariant as well.

The representation given above is not the only way to represent nominal sets. For example, first-order definable sets can be used as well (Klin & Szynwelski, 2016). From now on we assume to have set theoretic primitives so that each line in Algorithm 5.1 is well defined.

3.1 *Correctness*

To prove correctness we only have to prove that the algorithm terminates, that is, only finitely many hypotheses will be produced. Correctness follows trivially from termination since the last step of the algorithm is an equivalence query to the teacher inquiring whether an hypothesis automaton accepts the target language. We start out by listing some facts about observation tables.

Lemma 3. The relation $\sim$ is an equivariant equivalence relation. Furthermore, for all $u, v \in S$ we have that $u \equiv_{\mathcal{L}} v$ implies $u \sim v$.

This lemma implies that at any stage of the algorithm the number of orbits of $S/\sim$ does not exceed the number of orbits of the minimal acceptor with state space $A^*/\equiv_{\mathcal{L}}$ (recall that $\equiv_{\mathcal{L}}$ is the nominal Myhill-Nerode equivalence relation). Moreover, the following lemma shows that the dimension of the state space never exceeds the dimension of the minimal acceptor. Recall that the dimension is the maximal size of the support of any state, which is different than the number of orbits.

Lemma 4. We have $supp([u]_\sim) \subseteq supp([u]_{\equiv_{\mathcal{L}}}) \subseteq supp(u)$ for all $u \in S$.

Lemma 5. The automaton constructed from a closed and consistent table is minimal.

Proof. Follows from the categorical perspective by Jacobs and Silva (2014). □

We note that the constructed automaton is consistent with the table (we use that the set S is prefix-closed and E is suffix-closed (Angluin, 1987)). The following lemma shows that there are no strictly "smaller" automata consistent with the table. So the automaton is not just minimal, it is minimal w.r.t. the table.

Lemma 6. Let H be the automaton associated with a closed and consistent table (S, E). If M' is an automaton consistent with (S, E) (meaning that $se \in \mathcal{L}(M') \iff se \in \mathcal{L}(H)$ for all $s \in S \cup S{\cdot}A$ and $e \in E$) and M' has at most as many orbits as H, then there is a surjective map $f\colon Q_{M'} \to Q_H$. If moreover

- M''s dimension is bounded by the dimension of H, i.e., $\mathrm{supp}(m) \subseteq \mathrm{supp}(f(m))$ for all $m \in Q'_M$, and
- M' has no fewer local symmetries than H, i.e., $\pi \cdot f(m) = f(m)$ implies $\pi \cdot m = m$ for all $m \in Q'_M$,

then f defines an isomorphism $M' \cong H$ of nominal DFAs.

Proof. (All maps in this proof are equivariant.) Define a map $\mathrm{row}'\colon Q'_M \to 2^E$ by restricting the language map $Q'_M \to 2^{A^*}$ to E. First, observe that $\mathrm{row}'(\delta'(q'_0, s)) = \mathrm{row}(s)$ for all $s \in S \cup S{\cdot}A$, since $\epsilon \in E$ and M' is consistent with the table. Second, we have $\{\mathrm{row}'(\delta'(q'_0, s)) \mid s \in S\} \subseteq \{\mathrm{row}'(q) \mid q \in M'\}$.

Let n be the number of orbits of H. The former set has n orbits by the first observation, the latter set has at most n orbits by assumption. We conclude that the two sets (both being equivariant) must be equal. That means that for each $q \in M'$ there is a $s \in S$ such that $\mathrm{row}'(q) = \mathrm{row}(s)$. We see that $\mathrm{row}'\colon M' \to \{\mathrm{row}'(\delta'(q'_0, s)) \mid s \in S\} = H$ is a surjective map. Since a surjective map cannot increase the dimensions of orbits and the dimensions of M' are bounded, we note that the dimensions of the orbits in H and M' have to agree. Similarly, surjective maps preserve local symmetries. This map must hence be an isomorphism of nominal sets. Note that $\mathrm{row}'(q) = \mathrm{row}'(\delta'(q'_0, s))$ implies $q = \delta'(q'_0, s)$.

It remains to prove that it respects the automaton structures. It preserve the initial state: $\mathrm{row}'(q'_0) = \mathrm{row}(\delta'(q'_0, \epsilon)) = \mathrm{row}(\epsilon)$. Now let $q \in M'$ be a state and $s \in S$ such that $\mathrm{row}'(q) = \mathrm{row}(s)$. It preserves final states: $q \in F' \iff \mathrm{row}'(q)(\epsilon) = 1 \iff \mathrm{row}(s)(\epsilon) = 1$. Finally, it preserves the transition structure:

$$\mathrm{row}'(\delta'(q, a)) = \mathrm{row}'(\delta'(\delta'(q'_0, s), a)) = \mathrm{row}'(\delta'(q'_0, sa)) = \mathrm{row}(sa) = \delta(\mathrm{row}(s), a)$$

□

The above proof is an adaptation of Angluin's proof for automata over sets. We will now prove termination of the algorithm by proving that all steps are productive.

Theorem 7. The algorithm terminates and is hence correct.

Proof. Provided that the if-statements and set operations terminate, we are left proving that the algorithm adds (orbits of) rows and columns only finitely often. We start by proving that a table can be made closed and consistent in finite time.

If the table is not closed, we find a row $s_1 \in S{\cdot}A$ such that $\mathrm{row}(s_1) \neq \mathrm{row}(s)$ for all $s \in S$. The algorithm then adds the orbit containing s_1 to S. Since s_1 was nonequivalent to all rows, we find that $S \cup \mathrm{orb}(t)/{\sim}$ has strictly more orbits than $S/{\sim}$. Since orbits of $S/{\sim}$ cannot be more than those of $A^*/{\equiv_{\mathcal{L}}}$, this happens finitely often.

Columns are added in case of an inconsistency. Here the algorithm finds two elements $s_1, s_2 \in S$ with $row(s_1) = row(s_2)$ but $row(s_1 a e) \neq row(s_2 a e)$ for some $a \in A$ and $e \in E$. Adding ae to E will ensure that $row'(s_1) \neq row'(s_2)$ (row' is the function belonging to the updated observation table). If the two elements $row'(s_1), row'(s_2)$ are in different orbits, the number of orbits is increased. If they are in the same orbit, we have $row'(s_2) = \pi \cdot row'(s_1)$ for some permutation π. Using $row(s_1) = row(s_2)$ and $row'(s_1) \neq row'(s_2)$ we have:

$$row(s_1) = \pi \cdot row(s_1) \qquad row'(s_1) \neq \pi \cdot row'(s_1)$$

Consider all such π and suppose there is a π and $x \in supp(row(s_1))$ such that $\pi \cdot x \notin supp(row(s_1))$. Then we find that $\pi \cdot x \in supp(row'(s_1))$, and so the support of the row has grown. By Lemma 4 this happens finitely often. Suppose such π and x do not exist, then we consider the finite group $R = \{\rho|_{supp([s_1]_\sim)} \mid row(s_1) = \rho \cdot row(s_1)\}$. We see that $\{\rho|_{supp([s_1]_\sim)} \mid row'(s_1) = \rho \cdot row'(s_1)\}$ is a proper subgroup of R. So, adding a column in this case decreases the size of the group R, which can happen only finitely often. In this case a local symmetry is removed.

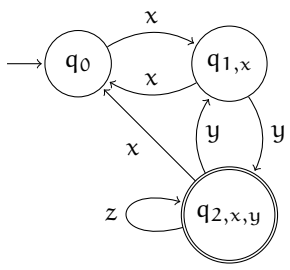
In short, the algorithm will succeed in producing a hypothesis in each round. It remains to prove that it needs only finitely many equivalence queries.

Let (S, E) be the closed and consistent table and H its corresponding hypothesis. If it is incorrect, then a second hypothesis H' will be constructed which is consistent with the old table (S, E). The two hypotheses are nonequivalent, as H' will handle the counterexample correctly and H does not. Therefore, H' will have at least one orbit more, one local symmetry less, or one orbit will have strictly bigger dimension (Lemma 6), all of which can only happen finitely often. □

We remark that all the lemmas and proofs as above are close to the original ones of Angluin. However, two things are crucially different. First, adding a column does not always increase the number of (orbits of) states. It can happen that by adding a column a bigger support is found or that a local symmetry is broken. Second, the new hypothesis does not necessarily have more states, again it might have bigger dimensions or less local symmetries.

From the proof Theorem 7 we observe moreover that the way we handle counterexamples is not crucial. Any other method which ensures a nonequivalent hypothesis will work. In particular our algorithm is easily adapted to include optimisations such as the ones by Maler and Pnueli (1995) and Rivest and Schapire (1993), where counterexamples are added as columns.[17]

[17] The additional optimisation of omitting the consistency check (Rivest & Schapire, 1993) cannot be done: we always add a whole orbit to S (to keep the set equivariant) and inconsistencies can arise within an orbit.



T_1	ϵ
ϵ	0
a	0
ab	1
aa	0
aba	0
abb	0
abc	1

T_2	ϵ	a'
ϵ	0	0
a	0	$\begin{cases} 1 & \text{if } a' \neq a \\ 0 & \text{else} \end{cases}$
ab	1	$\begin{cases} 1 & \text{if } a' \neq a,b \\ 0 & \text{else} \end{cases}$
aa	0	0
aba	0	0
abb	0	$\begin{cases} 1 & \text{if } a' \neq a \\ 0 & \text{else} \end{cases}$
abc	1	$\begin{cases} 1 & \text{if } a' \neq a,b \\ 0 & \text{else} \end{cases}$

T_3	ϵ	a'	$b'a'$
ϵ	0	0	1
a	0	$\begin{cases} 1 & \text{if } a' \neq a \\ 0 & \text{else} \end{cases}$	$\begin{cases} 1 & \text{if } a \neq a',b' \\ 0 & \text{else} \end{cases}$
ab	1	$\begin{cases} 1 & \text{if } a' \neq a,b \\ 0 & \text{else} \end{cases}$	$\begin{cases} 1 & \text{if } (b' \neq a,b \wedge a' \neq a,b) \vee (b' = b \wedge a' \neq a) \\ 0 & \text{else} \end{cases}$
aa	0	0	1
aba	0	0	1
abb	0	$\begin{cases} 1 & \text{if } a' \neq a \\ 0 & \text{else} \end{cases}$	$\begin{cases} 1 & \text{if } a \neq a',b' \\ 0 & \text{else} \end{cases}$
abc	1	$\begin{cases} 1 & \text{if } a' \neq a,b \\ 0 & \text{else} \end{cases}$	$\begin{cases} 1 & \text{if } (b' \neq a,b \wedge a' \neq a,b) \vee (b' = b \wedge a' \neq a) \\ 0 & \text{else} \end{cases}$

Figure 5.1 Example automaton to be learnt and three subsequent tables computed by νL^*. In the automaton, x, y, z denote distinct atoms.

3.2 *Example*

Consider the target automaton in Figure 5.1 and an observation table T_1 at some stage during the algorithm. We remind the reader that the table is represented in a symbolic way: The sequences in the rows and columns stand for whole orbits and the cells denote functions from the product of the orbits to 2. Since the cells can consist of multiple orbits, where each orbit is allowed to have a different value, we use a formula to specify which orbits have a 1.

The table T_1 has to be checked for closedness and consistency. We note that it is definitely closed. For consistency we check the rows $row(\epsilon)$ and $row(a)$ which are equal. Observe, however, that $row(\epsilon b)(\epsilon) = 0$ and $row(ab)(\epsilon) = 1$, so we have an inconsistency. The algorithm adds the orbit $orb(b)$ as column and extends the table, obtaining T_2. We note that, in this process, the number of orbits did grow, as the two rows are split. Furthermore, we see that both $row(a)$ and $row(ab)$ have empty support in T_1, but not in T_2, because $row(a)(a')$ depends on a' being equal or different from a, similarly for $row(ab)(a')$.

The table T_2 is still not consistent as we see that $row(ab) = row(ba)$ but $row(abb)(c) = 1$ and $row(bab)(c) = 0$. Hence the algorithm adds the columns $orb(bc)$, obtaining table T_3. We note that in this case, no new orbits are obtained and no support has grown. In fact, the only change here is that the local symmetry between $row(ab)$ and $row(ba)$ is removed. This last table, T_3, is closed and consistent and will produce the correct hypothesis.

3.3 *Query Complexity*

In this section, we will analyse the number of queries made by the algorithm in the worst case. Let M be the minimal target automaton with n orbits and of dimension k. We will use $\log$ in base two.

Lemma 8. The number of equivalence queries $E_{n,k}$ is $\mathcal{O}(nk \log k)$.

Proof. By Lemma 6 each hypothesis will be either 1) bigger in the number of orbits, which is bounded by n, or 2) bigger in the dimension of an orbit, which is bounded by k or 3) smaller in local symmetries of an orbit. For the last part we want to know how long a subgroup series of the permutation group S_k can be. This is bounded by the number of divisors of $k!$, as each subgroup divides the order of the group. We can easily bound the number of divisors of any m by $\log m$ and so one can at take a subgroup at most $k \log k$ times when starting with S_k.[18]

[18] After publication we found a better bound by Cameron, et al. (1989): the length of the longest chain of subgroups of S_k is $\left\lceil \frac{3}{2}k \right\rceil - b(k) - 1$, where $b(k)$ is the number of ones in the binary representation of k. This gives a linear bound in k, instead of the 'linearithmic' bound.

Since the hypothesis will grow monotonically in the number of orbits and for each orbit will grow monotonically w.r.t. the remaining two dimensions, the number of equivalence queries is bound by $n + n(k + k \log k)$. □

Next we will give a bound for the size of the table.

Lemma 9. The table has at most $n + mE_{n,k}$ orbits in S with sequences of at most length $n+m$, where m is the length of the longest counter example given by the teacher. The table has at most $n(k + k \log k + 1)$ orbits in E of at most length $n(k + k \log k + 1)$

Proof. In the termination proof we noted that rows are added at most n times. In addition (all prefixes of) counter examples are added as rows which add another $mE_{n,k}$ rows. Obviously counter examples are of length at most m and are extended at most n times, making the length at most $m + n$ in the worst case.

For columns we note that one of three dimensions approaches a bound similarly to the proof of Lemma 8. So at most $n(k + k \log k + 1)$ columns are added. Since they are suffix closed, the length is at most $n(k + k \log k + 1)$. □

Let p and l denote respectively the dimension and the number of orbits of A.

Lemma 10. The number of orbits in the lower part of the table, $S{\cdot}A$, is bounded by $(n + mE_{n,k})lf_{\mathbb{A}}(p(n + m), p)$.

Proof. Any sequence in S is of length at most $n + m$, so it contains at most $p(n + m)$ distinct atoms. When we consider $S{\cdot}A$, the extension can either reuse atoms from those $p(n+m)$, or none at all. Since the extra letter has at most p distinct atoms, the set $\mathbb{A}^{(p(n+m))} \times \mathbb{A}^{(p)}$ gives a bound $f_{\mathbb{A}}(p(n + m), p)$ for the number of orbits of $O_S \times O_A$, with O_X an orbit of X. Multiplying by the number of such ordered pairs, namely $(n + mE_{n,k})l$, gives a bound for $S{\cdot}A$. □

Let $C_{n,k,m} = (n+mE_{n,k})(lf_{\mathbb{A}}(p(n+m), p)+1)n(k+k \log k+1)$ be the maximal number of cells in the table. We note that this number is polynomial in k, l, m and n but it is not polynomial in p.

Corollary 11. The number of membership queries is bounded by

$$C_{n,k,m} f_{\mathbb{A}}(p(n + m), pn(k + k \log k + 1)).$$

4 Learning Non-Deterministic Nominal Automata

In this section, we introduce a variant of νL*, which we call νNL*, where the learnt automaton is non-deterministic. It will be based on the NL* algorithm by Bollig, et al. (2009), an Angluin-style algorithm for learning NFAs. The algorithm is shown

in Algorithm 5.2. We first illustrate NL*, then we discuss its extension to nominal automata.

NL* crucially relies on the use of *residual finite-state automata* (RFSA) (Denis, et al., 2002), which are NFAs admitting *unique minimal canonical representatives*. The states of this automaton correspond to Myhill-Nerode right-congruence classes, but can be exponentially smaller than the corresponding minimal DFA: *Composed* states, language-equivalent to sets of other states, can be dropped.

```
1   S, E ← {ε}
2   repeat
3     while (S, E) is not RFSA-closed or not RFSA-consistent do
4       if (S, E) is not RFSA-closed then
5         find s ∈ S, a ∈ A such that row(sa) ∈ PR(S, E) \ PR^⊤(S, E)
6         S ← S ∪ {sa}
7       end if
8       if (S, E) is not RFSA-consistent then
9         find s_1, s_2 ∈ S, a ∈ A, and e ∈ E such that
10                        row(s_1) ⊑ row(s_2) and L(s_1 ae) = 1, L(s_2 ae) = 0
11        E ← E ∪ {ae}
12      end if
13    end while
14    Make the conjecture N(S, E)
15    if the Teacher replies no, with a counter-example t then
16      E ← E ∪ suff(t)
17    end if
18  until the Teacher replies yes to the conjecture N(S, E)
19  return N(S, E)
```

Algorithm 5.2 Algorithm for learning NFAs by Bollig, et al. (2009).

The algorithm NL* equips the observation table (S, E) with a union operation, allowing for the detection of *composed* and *prime* rows.

Definition 12. Let $(row(s_1) \sqcup row(s_2))(e) = row(s_1)(e) \vee row(s_2)(e)$ (regarding cells as booleans). This operation induces an ordering between rows: $row(s_1) \sqsubseteq row(s_2)$ whenever $row(s_1)(e) = 1$ implies $row(s_2)(e) = 1$, for all $e \in E$.

A row $row(s)$ is *composed* if $row(s) = row(s_1) \sqcup \cdots \sqcup row(s_n)$, for $row(s_i) \neq row(s)$. Otherwise it is *prime*. We denote by $PR^\top(S, E)$ the rows in the top part of the table (ranging over S) which are prime w.r.t. the whole table (not only w.r.t. the top part). We write $PR(S, E)$ for all the prime rows of (S, E).

As in L^*, states of hypothesis automata will be rows of (S, E) but, as the aim is to construct a minimal RFSA, only prime rows are picked. New notions of closedness and consistency are introduced, to reflect features of RFSAs.

Definition 13. A table (S, E) is:
- *RFSA-closed* if, for all $t \in S \cdot A$, $row(t) = \bigsqcup\{row(s) \in PR^{\top}(S, E) \mid row(s) \sqsubseteq row(t)\}$;
- *RFSA-consistent* if, for all $s_1, s_2 \in S$ and $a \in A$, $row(s_1) \sqsubseteq row(s_2)$ implies $row(s_1 a) \sqsubseteq row(s_2 a)$.

If (S, E) is not RFSA-closed, then there is a row in the bottom part of the table which is prime, but not contained in the top part. This row is then added to S (line 5). If (S, E) is not RFSA-consistent, then there is a suffix which does not preserve the containment of two existing rows, so those rows are actually incomparable. A new column is added to distinguish those rows (line 10). Notice that counterexamples supplied by the teacher are added *to columns* (line 16). Indeed, it is shown by Bollig, et al. (2009) that treating the counterexamples as in the original L^*, namely adding them to rows, does not lead to a terminating algorithm.

Definition 14. Given a RFSA-closed and RFSA-consistent table (S, E), the conjecture automaton is $N(S, E) = (Q, Q_0, F, \delta)$, where:
- $Q = PR^{\top}(S, E)$;
- $Q_0 = \{r \in Q \mid r \sqsubseteq row(\epsilon)\}$;
- $F = \{r \in Q \mid r(\epsilon) = 1\}$;
- the transition relation is given by $\delta(row(s), a) = \{r \in Q \mid r \sqsubseteq row(sa)\}$.

As observed by Bollig, et al. (2009), $N(S, E)$ is not necessarily a RFSA, but it is a canonical RFSA if it is consistent with (S, E). If the algorithm terminates, then $N(S, E)$ must be consistent with (S, E), which ensures correctness. The termination argument is more involved than that of L^*, but still it relies on the minimal DFA.

Developing an algorithm to learn nominal NFAs is not an obvious extension of NL^*: Non-deterministic nominal languages strictly contain nominal regular languages, so it is not clear what the developed algorithm should be able to learn. To deal with this, we introduce a nominal notion of RFSAs. They are a *proper subclass* of nominal NFAs, because they recognise nominal regular languages. Nonetheless, they are more succinct than nominal DFAs.

4.1 Nominal Residual Finite-State Automata

Let $\mathcal{L}$ be a nominal language and u be a finite string. The *derivative* of $\mathcal{L}$ w.r.t. u is

$$u^{-1}\mathcal{L} = \{v \in A^* \mid uv \in \mathcal{L}\}.$$

A language $\mathcal{L}' \subseteq \mathbb{A}^*$ is a *residual* of $\mathcal{L}$ if there is u with $\mathcal{L}' = u^{-1}\mathcal{L}$. Note that a residual might not be equivariant, but it does have a finite support. We write $R(\mathcal{L})$ for the set of

residuals of $\mathcal{L}$. Residuals form an orbit-finite nominal set: They are in bijection with the state-space of the minimal nominal DFA for $\mathcal{L}$.

A *nominal residual finite-state automaton* for $\mathcal{L}$ is a nominal NFA whose states are subsets of such minimal automaton. Given a state q of an automaton, we write $\mathcal{L}(q)$ for the set of words leading from q to a set of states containing a final one.

Definition 15. A *nominal residual finite-state automaton* (nominal RFSA) is a nominal NFA $\mathcal{A}$ such that $\mathcal{L}(q) \in \mathtt{R}(\mathcal{L}(\mathcal{A}))$, for all $q \in Q_{\mathcal{A}}$.

Intuitively, all states of a nominal RSFA recognise residuals, but not all residuals are recognised by a single state: There may be a residual $\mathcal{L}'$ and a set of states Q' such that $\mathcal{L}' = \cup_{q \in Q'} \mathcal{L}(q)$, but no state q' is such that $\mathcal{L}(q') = \mathcal{L}'$. A residual $\mathcal{L}'$ is called *composed* if it is equal to the union of the components it strictly contains, explicitly

$$\mathcal{L}' = \bigcup \{\mathcal{L}'' \in \mathtt{R}(\mathcal{L}) \mid \mathcal{L}'' \subsetneq \mathcal{L}'\};$$

otherwise it is called *prime*. In an ordinary RSFA, composed residuals have finitely-many components. This is not the case in a nominal RFSA. However, the set of components of $\mathcal{L}'$ always has a finite support, namely $\mathtt{supp}(\mathcal{L}')$.

The set of prime residuals $\mathtt{PR}(\mathcal{L})$ is an orbit-finite nominal set, and can be used to define a *canonical* nominal RFSA for $\mathcal{L}$, which has the minimal number of states and the maximal number of transitions. This can be regarded as obtained from the minimal nominal DFA, by removing composed states and adding all initial states and transitions that do not change the recognised language. This automaton is necessarily unique.

Lemma 16. Let the *canonical* nominal RSFA of $\mathcal{L}$ be (Q, Q_0, F, δ) such that:
- $Q = \mathtt{PR}(\mathcal{L})$;
- $Q_0 = \{\mathcal{L}' \in Q \mid \mathcal{L}' \subseteq \mathcal{L}\}$;
- $F = \{\mathcal{L}' \in Q \mid \epsilon \in \mathcal{L}'\}$;
- $\delta(\mathcal{L}_1, a) = \{\mathcal{L}_2 \in Q \mid \mathcal{L}_2 \subseteq a^{-1}\mathcal{L}_1\}$.

It is a well-defined nominal NFA accepting $\mathcal{L}$.

4.2 νNL^*

Our nominal version of $\mathtt{NL}^*$ again makes use of an observation table (S, E) where S and E are equivariant subsets of A^* and $\mathtt{row}$ is an equivariant function. As in the basic algorithm, we equip (S, E) with a union operation $\sqcup$ and row containment relation $\sqsubseteq$, defined as in Definition 12. It is immediate to verify that $\sqcup$ and $\sqsubseteq$ are equivariant.

Our algorithm is a simple modification of the algorithm in Algorithm 5.2, where a few lines are replaced:

$6'$ $\quad S \leftarrow S \cup \mathrm{orb}(sa)$
$11'$ $\quad E \leftarrow E \cup \mathrm{orb}(ae)$
$16'$ $\quad E \leftarrow E \cup \mathrm{suff}(\mathrm{orb}(t))$

Switching to nominal sets, several decidability issues arise. The most critical one is that rows may be the union of infinitely many component rows, as happens for residuals of nominal languages, so finding all such components can be challenging. We adapt the notion of composed to rows: $\mathrm{row}(t)$ is composed whenever

$$\mathrm{row}(t) = \bigsqcup\{\mathrm{row}(s) \mid \mathrm{row}(s) \sqsubset \mathrm{row}(t)\}.$$

where $\sqsubset$ is *strict* row inclusion; otherwise $\mathrm{row}(t)$ is prime.

We now check that three relevant parts of our algorithm terminate.

1. Row containment check. The basic containment check $\mathrm{row}(s) \sqsubseteq \mathrm{row}(t)$ is decidable, as $\mathrm{row}(s)$ and $\mathrm{row}(t)$ are supported by the finite supports of s and t respectively.

2. RFSA-Closedness and RFSA-Consistency Checks. (Line 3)

We first show that prime rows form orbit-finite nominal sets.

Lemma 17. $\mathrm{PR}(S,E)$, $\mathrm{PR}^{\top}(S,E)$ and $\mathrm{PR}(S,E) \setminus \mathrm{PR}^{\top}(S,E)$ are orbit-finite nominal sets.

Consider now RFSA-closedness. It requires computing the set $C(\mathrm{row}(t))$ of components of $\mathrm{row}(t)$ contained in $\mathrm{PR}^{\top}(S,E)$ (possibly including $\mathrm{row}(t)$). This may not be equivariant under permutations $\mathrm{Perm}(\mathbb{A})$, but it is if we pick a subgroup.

Lemma 18. The set $C(\mathrm{row}(t))$ has the following properties:

- $\mathrm{supp}(C(\mathrm{row}(t))) \subseteq \mathrm{supp}(\mathrm{row}(t))$.
- it is equivariant and orbit-finite under the action of the group

$$G_t = \{\pi \in \mathrm{Perm}(\mathbb{A}) \mid \pi|_{\mathrm{supp}(\mathrm{row}(t))} = \mathrm{id}\}$$

  of permutations fixing $\mathrm{supp}(\mathrm{row}(t))$.

We established that $C(\mathrm{row}(t))$ can be effectively computed, and the same holds for $\bigsqcup C(\mathrm{row}(t))$. In fact, $\bigsqcup$ is equivariant w.r.t. the whole $\mathrm{Perm}(\mathbb{A})$ and then, in particular, w.r.t. G_t, so it preserves orbit-finiteness. Now, to check $\mathrm{row}(t) = \bigsqcup C(\mathrm{row}(t))$, we can just pick one representative of every orbit of $S \cdot A$, because we have $C(\pi \cdot \mathrm{row}(t)) = \pi \cdot C(\mathrm{row}(t))$ and permutations distribute over $\sqcup$, so permuting both sides of the equation gives again a valid equation.

For RFSA-consistency, consider the two sets

$$N = \{(s_1, s_2) \in S \times S \mid \mathrm{row}(s_1) \sqsubseteq \mathrm{row}(s_2)\}, \text{ and}$$
$$M = \{(s_1, s_2) \in S \times S \mid \forall a \in A : \mathrm{row}(s_1 a) \sqsubseteq \mathrm{row}(s_2 a)\}.$$

They are both orbit-finite nominal sets, by equivariance of row, $\sqsubseteq$ and A. We can check RFSA-consistency in finite time by picking orbit representatives from N and M. For each representative $n \in N$, we look for a representative $m \in M$ and a permutation π such that $n = \pi \cdot m$. If no such m and π exist, then n does not belong to any orbit of M, so it violates RFSA-consistency.

3. Finding Witnesses for Violations. (Lines 5 and 10) We can find witnesses by comparing orbit representatives of orbit-finite sets, as we did with RFSA-consistency. Specifically, we can pick representatives in $S \times A$ and $S \times S \times A \times E$ and check them against the following orbit-finite nominal sets:

- $\{(s, a) \in S \times A \mid row(sa) \in PR(S, E) \setminus PR^{\top}(S, E)\}$;
- $\{(s_1, s_2, a, e) \in S \times S \times A \times E \mid row(s_1 a)(e) = 1, row(s_2 a)(e) = 0, row(s_1) \sqsubseteq row(s_2)\}$;

4.3 *Correctness*

Now we prove correctness and termination of the algorithm. First, we prove that hypothesis automata are nominal NFAs.

Lemma 19. The hypothesis automaton $N(S, E)$ (see Definition 14) is a nominal NFA.

$N(S, E)$, as in ordinary $\mathtt{NL}^*$, is not always a nominal RFSA. However, we have the following.

Theorem 20. If the table (S, E) is RFSA-closed, RFSA-consistent and $N(S, E)$ is consistent with (S, E), then $N(S, E)$ is a canonical nominal RFSA.

This is proved by Bollig, et al. (2009) for ordinary RFSAs, using the standard theory of regular languages. The nominal proof is exactly the same, using derivatives of nominal regular languages and nominal RFSAs as defined in Section 4.1.

Lemma 21. The table (S, E) cannot have more than n orbits of distinct rows, where n is the number of orbits of the minimal nominal DFA for the target language.

Proof. Rows are residuals of $\mathcal{L}$, which are states of the minimal nominal DFA for $\mathcal{L}$, so orbits cannot be more than n. □

Theorem 22. The algorithm $\nu\mathtt{NL}^*$ terminates and returns the canonical nominal RFSA for $\mathcal{L}$.

Proof. If the algorithm terminates, then it must return the canonical nominal RFSA for $\mathcal{L}$ by Theorem 20. We prove that a table can be made RFSA-closed and RFSA-consistent in finite time. This is similar to the proof of Theorem 7 and is inspired by the proof of Theorem 2 of Bollig, et al. (2009).

If the table is not RFSA-closed, we find a row $s \in S{\cdot}A$ such that $row(s) \in PR(S,E) \setminus PR^{\top}(S,E)$. The algorithm then adds $orb(s)$ to S. Since s was nonequivalent to all upper prime rows, and thus from all the rows indexed by S, we find that $S \cup orb(t)/{\sim}$ has strictly more orbits than $S/{\sim}$ (recall that $s \sim t \iff row(s) = row(t)$). This addition can only be done finitely many times, because the number of orbits of $S/{\sim}$ is bounded, by Lemma 21.

Now, the case of RFSA-consistency needs some additional notions. Let R be the (orbit-finite) nominal set of all rows, and let $I = \{(r, r') \in R \times R \mid r \sqsubset r'\}$ be the set of all inclusion relations among rows. The set I is orbit-finite. In fact, consider

$$J = \{(s,t) \in (S \cup S{\cdot}A) \times (S \cup S{\cdot}A) \mid row(s) \sqsubset row(t)\}.$$

This set is an equivariant, thus orbit-finite, subset of $(S \cup S{\cdot}A) \times (S \cup S{\cdot}A)$. The set I is the image of J via $row \times row$, which is equivariant, so it preserves orbit-finiteness.

Now, suppose the algorithm finds two elements $s_1, s_2 \in S$ with $row(s_1) \sqsubseteq row(s_2)$ but $row(s_1 a)(e) = 1$ and $row(s_2 a)(e) = 0$ for some $a \in A$ and $e \in E$. Adding a column to fix RFSA-consistency may: **C1)** increase orbits of $(S \cup S{\cdot}A)/{\sim}$, or; **C2)** decrease orbits of I, or; **C3)** decrease local symmetries/increase dimension of one orbit of rows. In fact, if no new rows are added (**C1**), we have two cases.

- If $row(s_1) \sqsubset row(s_2)$, i.e., $(row(s_1), row(s_2)) \in I$, then $row'(s_1) \not\sqsubset row'(s_2)$, where row' is the new table. Therefore the orbit of $(row'(s_1), row'(s_2))$ is not in I. Moreover, $row'(s) \sqsubset row'(t)$ implies $row(s) \sqsubset row(t)$ (as no new rows are added), so no new pairs are added to I. Overall, I has less orbits (**C2**).
- If $row(s_1) = row(s_2)$, then we must have $row(s_1) = \pi \cdot row(s_1)$, for some π, because lines 4–7 forbids equal rows in different orbits. In this case $row'(s_1) \neq \pi \cdot row'(s_1)$ and we can use part of the proof of Theorem 7 to see that the orbit of $row'(s_1)$ has bigger dimension or less local symmetries than that of $row(s_1)$ (**C3**).

Orbits of $(S \cup S{\cdot}A)/{\sim}$ and of I are finitely-many, by Lemma 21 and what we proved above. Moreover, local symmetries can decrease finitely many times, and the dimension of each orbit of rows is bounded by the dimension of the minimal DFA state-space. Therefore all the above changes can happen finitely many times.

We have proved that the table eventually becomes RFSA-closed and RFSA-consistent. Now we prove that a finite number of equivalence queries is needed to reach the final hypothesis automaton. To do this, we cannot use a suitable version of Lemma 6, because this relies on $N(S,E)$ being consistent with (S,E), which in general is not true (see (Bollig, et al., 2008) for an example of this). We can, however, use an argument similar to that for RFSA-consistency, because the algorithm adds columns in response to counterexamples. Let w the counterexample provided by the teacher. When 16′ is executed, the table must change. In fact, by Lemma 2 of Bollig, et al. (2009), if it does not, then w is already correctly classified by $N(S,E)$, which is absurd. We have the following cases. **E1)** orbits of $(S \cup S{\cdot}A)/{\sim}$ increase (**C1**). Or, **E2)** either: Orbits in $PR(S,E)$ increase, or any of the following happens: Orbits in I decrease (**C2**), local symmetries/dimension of an orbit of rows change (**C3**). In fact, if **E1** does not happen

and $\mathtt{PR}(S, E)$, I and local symmetries/dimension of orbits of rows do not change, the automaton $\mathcal{A}$ for the new table coincides with $N(S, E)$. But $N(S, E) = \mathcal{A}$ is a contradiction, because $\mathcal{A}$ correctly classifies w (by Lemma 2 of Bollig, et al. (2009), as w now belongs to columns), whereas $N(S, E)$ does not. Both **E1** and **E2** can only happen finitely many times. □

4.4 *Query Complexity*

We now give bounds for the number of equivalence and membership queries needed by $\nu\mathtt{NL}^*$. Let n be the number of orbits of the minimal DFA M for the target language and let k be the dimension (i.e., the size of the maximum support) of its nominal set of states.

Lemma 23. The number of equivalence queries $E'_{n,k}$ is $O(n^2 f_{\mathbb{A}}(k, k) + nk \log k)$.

Proof. In the proof of Theorem 22, we saw that equivalence queries lead to more orbits in $(S \cup S{\cdot}A)/{\sim}$, in $\mathtt{PR}(S, E)$, less orbits in I or less local symmetries/bigger dimension for an orbit. Clearly the first two ones can happen at most n times. We now estimate how many times I can decrease. Suppose $(S \cup S{\cdot}A)/{\sim}$ has d orbits and h orbits are added to it. Recall that, given an orbit O of rows of dimension at most m, $f_{\mathbb{A}}(m, m)$ is an upper bound for the number of orbits in the product $O \times O$. Since the support of rows is bounded by k, we can give a bound for the number of orbits added to I: $dh f_{\mathbb{A}}(k, k)$, for new pairs $r \sqsubset r'$ with r in a new orbit of rows and r' in an old one (or vice versa); plus $(h(h-1)/2) f_{\mathbb{A}}(k, k)$, for r and r' both in (distinct) new orbits; plus $h f_{\mathbb{A}}(k, k)$, for r and r' in the same new orbit. Notice that, if $\mathtt{PR}(S, E)$ grows but $(S \cup S{\cdot}A)/{\sim}$ does not, I does not increase. By Lemma 21, $h, d \leqslant n$, so I cannot decrease more than $(n^2 + n(n-1)/2 + n) f_{\mathbb{A}}(k, k)$ times.

Local symmetries of an orbit of rows can decrease at most $k \log k$ times (see proof of Lemma 8), and its dimension can increase at most k times. Therefore $n(k + \log k)$ is a bound for all the orbits of rows, which are at most n, by Lemma 21. Summing up, we get the main result. □

Lemma 24. Let m be the length of the longest counterexample given by the teacher. Then the table has:

- at most n orbits in S, with words of length at most n;
- at most $mE'_{n,k}$ orbits in E, with words of length at most $mE'_{n,k}$.

Proof. By Lemma 21, the number of orbits of rows indexed by S is at most n. Now, notice that line 5 does not add $\mathtt{orb}(sa)$ to S if $sa \in S$, and lines 16 and 11 cannot identify rows, so S has at most n orbits. The length of the longest word in S must be at most n, as $S = \{\epsilon\}$ when the algorithm starts, and line 6′ adds words with one additional symbol than those in S.

For columns, we note that both fixing RFSA-consistency and adding counterexamples increase the number of columns, but this can happen at most $E'_{n,k}$ times (see proof of Lemma 23). Each time at most m suffixes are added to E. □

We compute the maximum number of cells as in Section 3.3.

Lemma 25. The number of orbits in the lower part of the table, $S{\cdot}A$, is bounded by $nlf_{\mathbb{A}}(pn, p)$.

Then $C'_{n,k,m} = n(lf_{\mathbb{A}}(pn, p) + 1)mE'_{n,k}$ is the maximal number of cells in the table. This bound is polynomial in n, m and l, but not in k and p.

Corollary 26. The number of membership queries is bounded by $C'_{n,k,m} f_{\mathbb{A}}(pn, pmE'_{n,k})$.

5 *Implementation and Preliminary Experiments*

Our algorithms for learning nominal automata operate on infinite sets of rows and columns, and hence it is not immediately clear how to actually implement them on a computer. We have used NLambda, a recently developed Haskell library by Klin and Szynwelski (2016) designed to allow direct manipulation of infinite (but orbit-finite) nominal sets, within the functional programming paradigm. The semantics of NLambda is based by Bojańczyk, et al. (2012), and the library itself is inspired by Fresh O'Caml by Shinwell (2006), a language for functional programming over nominal data structures with binding.

5.1 *NLambda*

NLambda extends Haskell with a new type `Atoms`. Values of this type are atomic values that can be compared for equality and have no other discernible structure. They correspond to the elements of the infinite alphabet $\mathbb{A}$ described in Section 2.

Furthermore, NLambda provides a unary type constructor `Set`. This appears similar to the the `Data.Set` type constructor from the standard Haskell library, but its semantics is markedly different: Whereas the latter is used to construct finite sets, the former has *orbit-finite* sets as values. The new constructor `Set` can be applied to a range of equality types that include `Atoms`, but also the tuple type `(Atoms, Atoms)`, the list type `[Atoms]`, the set type `Set Atoms`, and other types that provide basic infrastructure necessary to speak of supports and orbits. All these are instances of a type class `NominalType` specified in NLambda for this purpose.

NLambda, in addition to all the standard machinery of Haskell, offers primitives to manipulate values of any nominal types τ, σ:

– `empty` : `Set` τ, returns the empty set of any type;
– `atoms` : `Set Atoms`, returns the (infinite but single-orbit) set of all atoms;

- `insert` : $\tau \to$ `Set` $\tau \to$ `Set` τ, adds an element to a set;
- `map` : $(\tau \to \sigma) \to$ (`Set` $\tau \to$ `Set` σ), applies a function to every element of a set;
- `sum` : `Set` (`Set` τ) $\to$ `Set` τ, computes the union of a family of sets;
- `isEmpty` : `Set` $\tau \to$ `Formula`, checks whether a set is empty.

The type `Formula` has the role of a Boolean type. For technical reasons, it is distinct from the standard Haskell type `Bool`, but it provides standard logical operations, e.g.,

$$\texttt{not} : \texttt{Formula} \to \texttt{Formula}, \qquad \texttt{or} : \texttt{Formula} \to \texttt{Formula} \to \texttt{Formula},$$

as well as a conditional operator `ite` : `Formula` $\to \tau \to \tau \to \tau$ that mimics the standard `if-then-else` construction. It is also the result type of a built-in equality test on atoms:

$$\texttt{eq} : \texttt{Atoms} \to \texttt{Atoms} \to \texttt{Formula}.$$

Using these primitives, one builds more functions to operate on orbit-finite sets, such as a function to build singleton sets:

$$\begin{aligned} &\texttt{singleton} : \tau \to \texttt{Set}\ \tau \\ &\texttt{singleton}\ x = \texttt{insert}\ x\ \texttt{empty} \end{aligned}$$

or a filtering function to select elements that satisfy a given predicate:

$$\begin{aligned} &\texttt{filter} : (\tau \to \texttt{Formula}) \to \texttt{Set}\ \tau \to \texttt{Set}\ \tau \\ &\texttt{filter}\ p\ s = \texttt{sum}\ (\texttt{map}\ (\lambda x.\ \texttt{ite}\ (p\ x)\ (\texttt{singleton}\ x)\ \texttt{empty})\ s) \end{aligned}$$

or functions to quantify a predicate over a set:

$$\begin{aligned} &\texttt{exists}, \texttt{forall} : (\tau \to \texttt{Formula}) \to \texttt{Set}\ \tau \to \texttt{Formula} \\ &\texttt{exists}\ p\ s = \texttt{not}\ (\texttt{isEmpty}\ (\texttt{filter}\ p\ s)) \\ &\texttt{forall}\ p\ s = \texttt{isEmpty}\ (\texttt{filter}\ (\lambda x.\ \texttt{not}\ (p\ x))\ s) \end{aligned}$$

and so on. Note that these functions are written in exactly the same way as they would be for finite sets and the standard `Data.Set` type. This is not an accident, and indeed the programmer can use the convenient set-theoretic intuition of NLambda primitives. For example, one could conveniently construct various orbit-finite sets such as the set of all pairs of atoms:

$$\texttt{atomPairs} = \texttt{sum}\ (\texttt{map}\ (\lambda x.\ \texttt{map}\ (\lambda y.\ (x, y))\ \texttt{atoms})\ \texttt{atoms}),$$

the set of all pairs of *distinct atoms*:

$$\texttt{distPairs} = \texttt{filter}\ (\lambda(x, y).\ \texttt{not}\ (\texttt{eq}\ x\ y))\ \texttt{atomPairs}$$

and so on.

It should be stressed that all these constructions terminate in finite time, even though they formally involve infinite sets. To achieve this, values of orbit-finite set

types `Set` τ are internally not represented as lists or trees of elements of type τ. Instead, they are stored and manipulated symbolically, using first-order formulas over variables that range over atom values. For example, the value of `distPairs` above is stored as the formal expression:

$$\{(a, b) \mid a, b \in \mathbb{A},\ a \neq b\}$$

or, more specifically, as a triple:
- a pair (a, b) of "atom variables",
- a list $[a, b]$ of those atom variables that are bound in the expression (in this case, the expression contains no free variables),
- a formula $a \neq b$ over atom variables.

All the primitives listed above, such as `isEmpty`, `map` and `sum`, are implemented on this internal representation. In some cases, this involves checking the satisfiability of certain formulas over atoms. In the current implementation of NLambda, an external SMT solver Z3 (de Moura & Bjørner, 2008) is used for that purpose. For example, to evaluate the expression `isEmpty distPairs`, NLambda makes a system call to the SMT solver to check whether the formula $a \neq b$ is satisfiable in the first-order theory of equality and, after receiving the affirmative answer, returns the value `False`.

For more details about the semantics and implementation of NLambda, see Klin and Szynwelski (2016). The library itself can be downloaded from https://www.mimuw.edu.pl/~szynwelski/nlambda/.

5.2 *Implementation of νL^* and νNL^**

Using NLambda we implemented the algorithms from Sections 3 and 4. We note that the internal representation is slightly different than the one discussed in Section 3. Instead of representing the table (S, E) with actual representatives of orbits, the sets are represented logically as described above. Furthermore, the control flow of the algorithm is adapted to fit in the functional programming paradigm. In particular, recursion is used instead of a while loop. In addition to the nominal adaptation of Angluin's algorithm νL^*, we implemented a variant, νL^*_{col} which adds counterexamples to the columns instead of rows.

Target automata are defined using NLambda as well, using the automaton data type provided by the library. Membership queries are already implemented by the library. Equivalence queries are implemented by constructing a bisimulation (recall that bisimulation implies language equivalence), where a counterexample is obtained when two DFAs are not bisimilar. For nominal NFAs, however, we cannot implement a complete equivalence query as their language equivalence is undecidable. We approximated the equivalence by bounding the depth of the bisimulation for nominal NFAs. As an optimisation, we use bisimulation up to congruence as described by Bonchi and Pous (2015). Having an approximate teacher is a minor issue since in

many applications no complete teacher can be implemented and one relies on testing (Aarts, et al., 2015 and Bollig, et al., 2013). For the experiments listed here the bound was chosen large enough for the learner to terminate with the correct automaton.

The code can be found at https://github.com/Jaxan/nominal-lstar.

5.3 *Test Cases*

To provide a benchmark for future improvements, we tested our algorithms on simple automata described below. We report results in Table 5.1. The experiments were performed on a machine with an Intel Core i5 (Skylake, 2.4 GHz) and 8 GB RAM.

Model			νL* (s)	νL*$_{\text{col}}$ (s)			νNL* (s)
FIFO_0	2	0	1.9	1.9	2	0	2.4
FIFO_1	3	1	12.9	7.4	3	1	17.3
FIFO_2	5	2	45.6	22.6	5	2	70.3
FIFO_3	10	3	189	107	10	3	476
FIFO_4	25	4	370	267	25	4	1230
FIFO_5	77	5	1337	697	∞	∞	∞
$\mathcal{L}_0$	2	0	1.3	1.4	2	0	1.4
$\mathcal{L}_1$	4	1	29.6	4.7	4	1	8.9
$\mathcal{L}_2$	7	2	229	23.1	7	2	84.7
$\mathcal{L}'_0$	3	1	4.4	4.9	3	1	11.3
$\mathcal{L}'_1$	5	1	15.4	15.4	4	1	66.4
$\mathcal{L}'_2$	9	1	46.3	40.5	5	1	210
$\mathcal{L}'_3$	17	1	89.0	66.8	6	1	566
$\mathcal{L}_{\text{eq}}$	n/a	n/a	n/a	n/a	3	1	16.3

Table 5.1 Results of experiments. The column DFA (resp. RFSA) shows the number of orbits (left sub-column) and dimension (right sub-column) of the learnt minimal DFA (resp. canonical RFSA). We use ∞ when the running time is too high.

Queue Data Structure. A queue is a data structure to store elements which can later be retrieved in a first-in, first-out order. It has two operations: `push` and `pop`. We define the alphabet $\Sigma_{\text{FIFO}} = \{\texttt{push}(a), \texttt{pop}(a) \mid a \in \mathbb{A}\}$. The language FIFO_n contains all valid traces of `push` and `pop` using a bounded queue of size n. The minimal nominal DFA for FIFO_2 is given in Figure 5.2.

The state reached from $q_{1,x}$ via $\texttt{push}(x)$ is omitted: Its outgoing transitions are those of $q_{2,x,y}$, where y is replaced by x. Similar benchmarks appear in (Aarts, et al., 2015 and Isberner, et al., 2014).

Double Word. $\mathcal{L}_n = \{ww \mid w \in \mathbb{A}^n\}$ from Section 1.

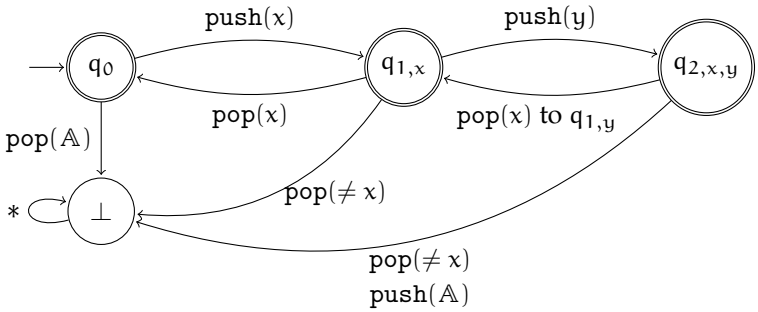


Figure 5.2 A nominal automaton accepting FIFO_2.

NFA. Consider the language $\mathcal{L}_{eq} = \cup_{a\in\mathbb{A}} \mathbb{A}^*a\mathbb{A}^*a\mathbb{A}^*$ of words where some letter appears twice. This is accepted by an NFA which guesses the position of the first occurrence of a repeated letter a and then waits for the second a to appear. The language is not accepted by a DFA (Bojańczyk, et al., 2014). Despite this $\nu\mathtt{NL}^*$ is able to learn the automaton shown in Figure 5.3.

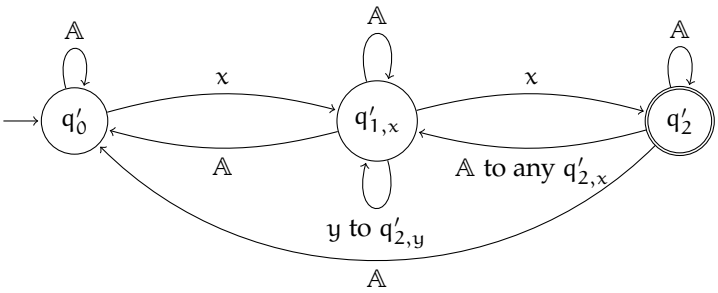


Figure 5.3 A nominal NFA accepting $\mathcal{L}_{eq}$. Here, the transition from q'_2 to $q'_{1,x}$ is defined as $\delta(q'_2, a) = \{q'_{1,b} \mid b \in \mathbb{A}\}$.

n-last Position. A prototypical example of regular languages which are accepted by very small NFAs is the set of words where a distinguished symbol a appears on the n-last position (Bollig, et al., 2009). We define a similar nominal language $\mathcal{L}'_n = \cup_{a\in\mathbb{A}} a\mathbb{A}^*a\mathbb{A}^n$. To accept such words non-deterministically, one simply guesses the n-last position. This language is also accepted by a much larger deterministic automaton.

6 Related Work

This section compares $\nu\mathtt{L}^*$ with other algorithms from the literature. We stress that no comparison is possible for $\nu\mathtt{NL}^*$, as it is the first learning algorithm for non-deterministic automata over infinite alphabets.

The first one to consider learning automata over infinite alphabets was Sakamoto (1997). In his work the problem is reduced to L^* with some finite sub-alphabet. The sub-alphabet grows in stages and L^* is rerun at every stage, until the alphabet is big enough to capture the whole language. In Sakamoto's approach, any learning algorithm can be used as a back-end. This, however, comes at a cost: It has to be rerun at every stage, and each symbol is treated in isolation, which might require more queries. Our algorithm νL^*, instead, works with the whole alphabet from the very start, and it exploits its symmetry. An example is in Sections 1.1 and 1.2: The ordinary learner uses four equivalence queries, whereas the nominal one, using the symmetry, only needs three. Moreover, our algorithm is easier to generalise to other alphabets and computational models, such as non-determinism.

More recently papers appeared on learning register automata by Cassel, et al. (2016) and Howar, et al. (2012). Their register automata are as expressive as our deterministic nominal automata. The state space is similar to our orbit-wise representation: It is formed by finitely many locations with registers. Transitions are defined symbolically using propositional logic. We remark that the most recent paper by Cassel, et al. (2016) generalises the algorithm to alphabets with different structures (which correspond to different atom symmetries in our work), but at the cost of changing Angluin's framework. Instead of membership queries the algorithm requires more sophisticated tree queries. In our approach, using a different symmetry does not affect neither the algorithm nor its correctness proof. Tree queries can be reduced to membership queries by enumerating all n-types for some n (n-types in logic correspond to orbits in the set of n-tuples). Keeping that in mind, their complexity results are roughly the same as ours, although this is hard to verify, as they do not give bounds on the length of individual tree queries. Finally, our approach lends itself better to be extended to other variations on L^* (of which many exist), as it is closer to Angluin's original work.

Another class of learning algorithms for systems with large alphabets is based on abstraction and refinement, which is orthogonal to the approach in this thesis but connections and possible transference of techniques are worth exploring in the future. Aarts, et al. (2015) reduce the alphabet to a finite alphabet of abstractions, and L^* for ordinary DFAs over such finite alphabet is used. Abstractions are refined by counterexamples. Other similar approaches are by Howar, et al. (2011) and Isberner, et al. (2013), where global and local per-state abstractions of the alphabet are used, and by Mens (2017), where the alphabet can also have additional structure (e.g., an ordering relation). We also mention that Botincan and Babic (2013) give a framework for learning symbolic models of software behaviour.

Berg, et al. (2006 and 2008) cope with an infinite alphabet by running L^* (adapted to Mealy machines) using a finite approximation of the alphabet, which may be augmented when equivalence queries are answered. A smaller symbolic model is derived subsequently. Their approach, unlike ours, does not exploit the symmetry

over the full alphabet. The symmetry allows our algorithm to reduce queries and to produce the smallest possible automaton at every step.

Finally we compare with results on session automata (Bollig, et al., 2013). Session automata are defined over finite alphabets just like the work by Sakamoto. However, session automata are more restrictive than deterministic nominal automata. For example, the model cannot capture an acceptor for the language of words where consecutive data values are distinct. This language can be accepted by a three orbit nominal DFA, which can be learned by our algorithm.

We implemented our algorithms in the nominal library NLambda as sketched before. Other implementation options include Fresh OCaml (Shinwell, 2006), a functional programming language designed for programming over nominal data structures with binding, and Lois by Kopczyński and Toruńczyk (2016 and 2017), a C++ library for imperative nominal programming. We chose NLambda for its convenient set-theoretic primitives, but the other options remain to be explored, in particular the low-level Lois could be expected to provide more efficient implementations.

7 *Discussion and Future Work*

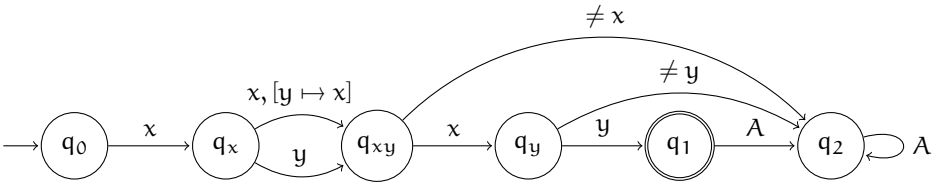
In this chapter we defined and implemented extensions of several versions of L* and of NL* for nominal automata.
We highlight two features of our approach:

- It has strong theoretical foundations: The *theory of nominal languages*, covering different alphabets and symmetries (see Section 2.1); *category theory*, where nominal automata have been characterised as *coalgebras* (Ciancia & Montanari, 2010 and Kozen, et al., 2015) and many properties and algorithms (e.g., minimisation) have been studied at this abstract level.
- It follows a generic pattern for transporting computation models and algorithms from finite sets to nominal sets, which leads to simple correctness proofs.

These features pave the way to several extensions and improvements.

Future work includes a general version of νNL*, parametric in the notion of side-effect (an example is non-determinism). Different notions will yield models with different degree of succinctness w.r.t. deterministic automata. The key observation here is that many forms of non-determinism and other side effects can be captured via the categorical notion of *monad*, i.e., an algebraic structure, on the state-space. Monads allow generalising the notion of composed and prime state: A state is composed whenever it is obtained from other states via an algebraic operation. Our algorithm νNL* is based on the *powerset* monad, representing classical non-determinism. We are currently investigating a *substitution* monad, where the operation is "applying a (possibly non-injective) substitution of atoms in the support". A minimal automaton over this monad, akin to a RFSA, will have states that can generate all the states of the associated minimal DFA via a substitution, but cannot be generated by other states

(they are prime). For instance, we can give an automaton over the substitution monad that recognises $\mathcal{L}_2$ from Section 1:



Here $[y \mapsto x]$ means that, if that transition is taken, q_{xy} (hence its language) is subject to $y \mapsto x$. In general, the size of the minimal DFA for $\mathcal{L}_n$ grows more than exponentially with n, but an automaton with substitutions on transitions, like the one above, only needs $\mathcal{O}(n)$ states. This direction is investigated in Chapter 7.

In principle, thanks to the generic approach we have taken, all our algorithms should work for various kinds of atoms with more structure than just equality, as advocated by Bojańczyk, et al. (2014). Details, such as precise assumptions on the underlying structure of atoms necessary for proofs to go through, remain to be checked. In the next chapter (Chapter 6), we investigate learning with the total order symmetry. We implement this in NLambda, as well as a new tool for computing with nominal sets over the total order symmetry.

The efficiency of our current implementation, as measured in Section 5.3, leaves much to be desired. There is plenty of potential for running time optimisation, ranging from improvements in the learning algorithms itself, to optimisations in the NLambda library (such as replacing the external and general-purpose SMT solver with a purpose-built, internal one, or a tighter integration of nominal mechanisms with the underlying Haskell language as it was done by Shinwell, 2006), to giving up the functional programming paradigm for an imperative language such as LOIS (Kopczyński & Toruńczyk, 2016 and 2017).

Acknowledgements

We thank Frits Vaandrager and Gerco van Heerdt for useful comments and discussions. We also thank the anonymous reviewers.

Chapter 6
Fast Computations on Ordered Nominal Sets

David Venhoek
Radboud University

Joshua Moerman
Radboud University

Jurriaan Rot
Radboud University

Abstract

We show how to compute efficiently with nominal sets over the total order symmetry by developing a direct representation of such nominal sets and basic constructions thereon. In contrast to previous approaches, we work directly at the level of orbits, which allows for an accurate complexity analysis. The approach is implemented as the library Ons (Ordered Nominal Sets).

Our main motivation is nominal automata, which are models for recognising languages over infinite alphabets. We evaluate Ons in two applications: minimisation of automata and active automata learning. In both cases, Ons is competitive compared to existing implementations and outperforms them for certain classes of inputs.

This chapter is based on the following publication:
Venhoek, D., Moerman, J., & Rot, J. (2018). Fast Computations on Ordered Nominal Sets. In *Theoretical Aspects of Computing - ICTAC - 15th International Colloquium, Proceedings*. Springer. doi:10.1007/978-3-030-02508-3_26

Automata over infinite alphabets are natural models for programs with unbounded data domains. Such automata, often formalised as *register automata*, are applied in modelling and analysis of communication protocols, hardware, and software systems (see Bojańczyk, et al., 2014; D'Antoni & Veanes, 2017; Grigore & Tzevelekos, 2016; Kaminski & Francez, 1994; Montanari & Pistore, 1997; Segoufin, 2006 and references therein). Typical infinite alphabets include sequence numbers, timestamps, and identifiers. This means one can model data flow in such automata beside the basic control flow provided by ordinary automata. Recently, it has been shown in a series of papers that such models are amenable to learning (Aarts, et al., 2015; Bollig, et al., 2013; Cassel, et al., 2016; Drews & D'Antoni, 2017; Moerman, et al., 2017; Vaandrager, 2017) with the verification of (closed source) TCP implementations by Fiterău-Broștean, et al. (2016) as a prominent example.

A foundational approach to infinite alphabets is provided by the notion of *nominal set*, originally introduced in computer science as an elegant formalism for name binding (Gabbay & Pitts, 2002 and Pitts, 2016). Nominal sets have been used in a variety of applications in semantics, computation, and concurrency theory (see Pitts, 2013 for an overview). Bojańczyk, et al. (2014) introduce *nominal automata*, which allow one to model languages over infinite alphabets with different symmetries. Their results are parametric in the structure of the data values. Important examples of data domains are ordered data values (e.g., timestamps) and data values that can only be compared for equality (e.g., identifiers). In both data domains, nominal automata and register automata are equally expressive.

Important for applications of nominal sets and automata are implementations. A couple of tools exist to compute with nominal sets. Notably, Nλ (Klin & Szynwelski, 2016) and Lois (Kopczyński & Toruńczyk, 2016 and 2017) provide a general purpose programming language to manipulate infinite sets.[19] Both tools are based on SMT solvers and use logical formulas to represent the infinite sets. These implementations are very flexible, and the SMT solver does most of the heavy lifting, which makes the implementations themselves relatively straightforward. Unfortunately, this comes at a cost as SMT solving is in general Pspace-hard. Since the formulas used to describe sets tend to grow as more calculations are done, running times can become unpredictable.

In this chapter, we use a direct representation based on symmetries and orbits, to represent nominal sets. We focus on the *total order symmetry*, where data values are rational numbers and can be compared for their order. Nominal automata over the total order symmetry are more expressive than automata over the equality symmetry (i.e., traditional register automata of Kaminski & Francez, 1994). A key insight is that the representation of nominal sets from Bojańczyk, et al. (2014) becomes rather simple in the total order symmetry; each orbit is presented solely by a natural number, intuitively representing the number of variables or registers.

[19] Other implementations of nominal techniques that are less directly related to our setting (Mihda, Fresh OCaml, and Nominal Isabelle) are discussed in Section 5.

Our main contributions include the following.

- We develop the *representation theory* of nominal sets over the total order symmetry. We give concrete representations of nominal sets, their products, and equivariant maps.
- We provide *time complexity bounds* for operations on nominal sets such as intersections and membership. Using those results we give the time complexity of Moore's minimisation algorithm (generalised to nominal automata) and prove that it is polynomial in the number of orbits.
- Using the representation theory, we are able to *implement nominal sets in a C++ library* Ons. The library includes all the results from the representation theory (sets, products, and maps).
- We *evaluate the performance* of Ons and compare it to Nλ and Lois, using two algorithms on nominal automata: minimisation (Bojańczyk & Lasota, 2012) and automata learning (Moerman, et al., 2017). We use randomly generated automata as well as concrete, logically structured models such as FIFO queues. For random automata, our methods are drastically faster than the other tools. On the other hand, Lois and Nλ are faster in minimising the structured automata as they exploit their logical structure. In automata learning, the logical structure is not available a-priori, and Ons is faster in most cases.

The structure of this chapter is as follows. Section 1 contains background on nominal sets and their representation. Section 2 describes the concrete representation of nominal sets, equivariant maps and products in the total order symmetry. Section 3 describes the implementation Ons with complexity results, and Section 4 the evaluation of Ons on algorithms for nominal automata. Related work is discussed in Section 5, and future work in Section 6.

1 Nominal sets

Nominal sets are infinite sets that carry certain symmetries, allowing a finite representation in many interesting cases. We recall their formalisation in terms of group actions, following Bojańczyk, et al. (2014) and Pitts (2013), to which we refer for an extensive introduction.

1.1 Group actions

Let G be a group and X be a set. A (*left*) G-*action* is a function $\cdot : G \times X \to X$ satisfying $1 \cdot x = x$ and $(hg) \cdot x = h \cdot (g \cdot x)$ for all $x \in X$ and $g, h \in G$. A set X with a G-action is called a G-*set* and we often write gx instead of $g \cdot x$. The *orbit* of an element $x \in X$ is the set $\{gx \mid g \in G\}$. A G-set is always a disjoint union of its orbits (in other words, the orbits partition the set). We say that X is *orbit-finite* if it has finitely many orbits, and we denote the number of orbits by $N(X)$.

A map $f\colon X \to Y$ between G-sets is called *equivariant* if it preserves the group action, i.e., for all $x \in X$ and $g \in G$ we have $g \cdot f(x) = f(g \cdot x)$. If an equivariant map f is bijective, then f is an *isomorphism* and we write $X \cong Y$. A subset $Y \subseteq X$ is equivariant if the corresponding inclusion map is equivariant. The *product* of two G-sets X and Y is given by the Cartesian product $X \times Y$ with the point-wise group action on it, i.e., $g(x, y) = (gx, gy)$. Union and intersection of X and Y are well-defined if the two actions agree on their common elements.

1.2 *Nominal sets*

A *data symmetry* is a pair $(\mathcal{D}, G)$ where $\mathcal{D}$ is a set and G is a subgroup of $\mathrm{Sym}(\mathcal{D})$, the group of bijections on $\mathcal{D}$. Note that the group G naturally acts on $\mathcal{D}$ by defining $gx = g(x)$. In the most studied instance, called the *equality symmetry*, $\mathcal{D}$ is a countably infinite set and $G = \mathrm{Sym}(\mathcal{D})$. In this chapter, we focus on the *total order symmetry* given by $\mathcal{D} = \mathbb{Q}$ and $G = \{\pi \mid \pi \in \mathrm{Sym}(\mathbb{Q}), \pi \text{ is monotone}\}$.

Let $(\mathcal{D}, G)$ be a data symmetry and X be a G-set. A set of data values $S \subseteq \mathcal{D}$ is called a *support* of an element $x \in X$ if for all $g \in G$ with $\forall s \in S : gs = s$ we have $gx = x$. A G-set X is called *nominal* if every element $x \in X$ has a finite support.

Example 1. We list several examples for the total order symmetry. The set $\mathbb{Q}^2$ is nominal as each element $(q_1, q_2) \in \mathbb{Q}^2$ has the finite set $\{q_1, q_2\}$ as its support. The set has the following three orbits:

$$\{(q_1, q_2) \mid q_1 < q_2\} \qquad \{(q_1, q_2) \mid q_1 = q_2\} \qquad \{(q_1, q_2) \mid q_1 > q_2\}.$$

For a set X, the set of all subsets of size $n \in \mathbb{N}$ is denoted by $\mathcal{P}_n(X) = \{Y \subseteq X \mid \#Y = n\}$. The set $\mathcal{P}_n(\mathbb{Q})$ is a single-orbit nominal set for each n, with the action defined by direct image: $gY = \{gy \mid y \in Y\}$. The group of monotone bijections also acts by direct image on the full power set $\mathcal{P}(\mathbb{Q})$, but this is *not* a nominal set. For instance, the set $\mathbb{Z} \in \mathcal{P}(\mathbb{Q})$ of integers has no finite support.

If $S \subseteq \mathcal{D}$ is a support of an element $x \in X$, then any set $S' \subseteq \mathcal{D}$ such that $S \subseteq S'$ is also a support of x. A set $S \subset \mathcal{D}$ is a *least finite support* of $x \in X$ if it is a finite support of x and $S \subseteq S'$ for any finite support S' of x. The existence of least finite supports is crucial for representing orbits. Unfortunately, even when elements have a finite support, in general they do not always have a least finite support. A data symmetry $(\mathcal{D}, G)$ is said to *admit least supports* if every element of every nominal set has a least finite support. Both the equality and the total order symmetry admit least supports. (Bojańczyk, et al., 2014 give additional (counter)examples of data symmetries admitting least supports.) Having least finite supports is useful for a finite representation. Henceforth, we will write *least support* to mean least finite support.

Given a nominal set X, the size of the least support of an element $x \in X$ is denoted by $\dim(x)$, the *dimension* of x. We note that all elements in the orbit of x have the same

dimension. For an orbit-finite nominal set X, we define $\dim(X) = \max\{\dim(x) \mid x \in X\}$. For a single-orbit set O, observe that $\dim(O) = \dim(x)$ where x is any element $x \in O$.

1.3 *Representing nominal orbits*

We represent nominal sets as collections of single orbits. The finite representation of single orbits is based on the theory of Bojańczyk, et al. (2014), which uses the technical notions of *restriction* and *extension*. We only briefly report their definitions here. However, the reader can safely move to the concrete representation theory in Section 2 with only a superficial understanding of Theorem 2 below.

The *restriction* of an element $\pi \in G$ to a subset $C \subseteq \mathcal{D}$, written as $\pi|_C$, is the restriction of the function $\pi\colon \mathcal{D} \to \mathcal{D}$ to the domain C. The restriction of a group G to a subset $C \subseteq \mathcal{D}$ is defined as $G|_C = \{\pi|_C \mid \pi \in G,\ \pi C = C\}$. The *extension* of a subgroup $S \leqslant G|_C$ is defined as $ext_G(S) = \{\pi \in G \mid \pi|_C \in S\}$. For $C \subseteq \mathcal{D}$ and $S \leqslant G|_C$, define $[C, S]^{ec} = \{\{gs \mid s \in ext_G(S)\} \mid g \in G\}$, i.e., the set of right cosets of $ext_G(S)$ in G. Then $[C, S]^{ec}$ is a single-orbit nominal set.

Using the above, we can formulate the representation theory from Bojańczyk, et al. (2014). This gives a finite description for all single-orbit nominal sets X, namely a finite set C together with some of its symmetries.

Theorem 2. Let X be a single-orbit nominal set for a data symmetry $(\mathcal{D}, G)$ that admits least supports and let $C \subseteq \mathcal{D}$ be the least support of some element $x \in X$. Then there exists a subgroup $S \leqslant G|_C$ such that $X \cong [C, S]^{ec}$.

The proof by Bojańczyk, et al. (2014) uses a bit of category theory: it establishes an equivalence of categories between single-orbit sets and the pairs (C, S). We will not use the language of category theory much in order to keep the chapter self-contained.

2 *Representation in the total order symmetry*

This section develops a concrete representation of nominal sets over the total order symmetry, as well as their equivariant maps and products. It is based on the abstract representation theory from Section 1.3. From now on, by *nominal set* we always refer to a nominal set over the total order symmetry. Hence, our data domain is $\mathbb{Q}$ and we take G to be the group of monotone bijections.

2.1 *Orbits and nominal sets*

From the representation in Section 1.3, we find that any single-orbit set X can be represented as a tuple (C, S). Our first observation is that the finite group S of 'local

symmetries' in this representation is always trivial, i.e., $S = I$, where $I = \{1\}$ is the trivial group. This follows from the following lemma and $S \leqslant G|_C$.

Lemma 3. For every finite subset $C \subset \mathbb{Q}$, we have $G|_C = I$.

Immediately, we see that $(C, S) = (C, I)$, and hence that the orbit is fully represented by the set C. A further consequence of Lemma 3 is that each *element* of an orbit can be uniquely identified by its least support. This leads us to the following characterisation of $[C, I]^{ec}$.

Lemma 4. Given a finite subset $C \subset \mathbb{Q}$, we have $[C, I]^{ec} \cong \mathcal{P}_{\#C}(\mathbb{Q})$.

By Theorem 2 and the above lemmas, we can represent an orbit by a single integer n, the size of the least support of its elements. This naturally extends to (orbit-finite) nominal sets with multiple orbits by using a multiset of natural numbers, representing the size of the least support of each of the orbits. These multisets are formalised here as functions $f\colon \mathbb{N} \to \mathbb{N}$.

Definition 5. Given a function $f\colon \mathbb{N} \to \mathbb{N}$, we define a nominal set $[f]^o$ by

$$[f]^o = \bigcup_{\substack{n \in \mathbb{N} \\ 1 \leqslant i \leqslant f(n)}} \{i\} \times \mathcal{P}_n(\mathbb{Q}).$$

Proposition 6. For every orbit-finite nominal set X, there is a function $f\colon \mathbb{N} \to \mathbb{N}$ such that $X \cong [f]^o$ and the set $\{n \mid f(n) \neq 0\}$ is finite. Furthermore, the mapping between X and f is one-to-one (up to isomorphism of nominal sets) when restricting to $f\colon \mathbb{N} \to \mathbb{N}$ for which the set $\{n \mid f(n) \neq 0\}$ is finite.

The presentation in terms of a function $f\colon \mathbb{N} \to \mathbb{N}$ enforces that there are only finitely many orbits of any given dimension. The first part of the above proposition generalises to arbitrary nominal sets by replacing the codomain of f by the class of all sets and adapting Definition 5 accordingly. However, the resulting correspondence will no longer be one-to-one.

As a brief example, let us consider the set $\mathbb{Q} \times \mathbb{Q}$. The elements (a, b) split in three orbits, one for $a < b$, one for $a = b$ and one for $a > b$. These have dimension 2, 1 and 2 respectively, so the set $\mathbb{Q} \times \mathbb{Q}$ is represented by the multiset $\{1, 2, 2\}$.

2.2 *Equivariant maps*

We show how to represent equivariant maps, using two basic properties. Let $f\colon X \to Y$ be an equivariant map. The first property is that the direct image of an orbit (in X) is again an orbit (in Y), that is to say, f is defined 'orbit-wise'. Second, equivariant maps cannot introduce new elements in the support (but they can drop them). More precisely:

Lemma 7. Let $f\colon X \to Y$ be an equivariant map, and $O \subseteq X$ a single orbit. The direct image $f(O) = \{f(x) \mid x \in O\}$ is a single-orbit nominal set.

Lemma 8. Let $f\colon X \to Y$ be an equivariant map between two nominal sets X and Y. Let $x \in X$ and let C be a support of x. Then C supports $f(x)$.

Hence, equivariant maps are fully determined by associating two pieces of information for each orbit in the domain: the orbit on which it is mapped and a string denoting which elements of the least support of the input are preserved. These ingredients are formalised in the first part of the following definition. The second part describes how these ingredients define an equivariant function. Proposition 10 then states that every equivariant function can be described in this way.

Definition 9. Let $H = \{(I_1, F_1, O_1), ..., (I_n, F_n, O_n)\}$ be a finite set of tuples where the I_i's are disjoint single-orbit nominal sets, the O_i's are single-orbit nominal sets with $\dim(O_i) \leqslant \dim(I_i)$, and the F_i's are bit strings of length $\dim(I_i)$ with exactly $\dim(O_i)$ ones.

Given a set H as above, we define $f_H\colon \cup I_i \to \cup O_i$ as the unique equivariant function such that, given $x \in I_i$ with least support C, $f_H(x)$ is the unique element of O_i with support $\{C(j) \mid F_i(j) = 1\}$, where $F_i(j)$ is the j-th bit of F_i and $C(j)$ is the j-th smallest element of C.

Proposition 10. For every equivariant map $f\colon X \to Y$ between orbit-finite nominal sets X and Y there is a set H as in Definition 9 such that $f = f_H$.

Consider the example function $\mathit{min} : \mathcal{P}_3(\mathbb{Q}) \to \mathbb{Q}$ which returns the smallest element of a 3-element set. Note that both $\mathcal{P}_3(\mathbb{Q})$ and $\mathbb{Q}$ are single orbits. Since for the orbit $\mathcal{P}_3(\mathbb{Q})$ we only keep the smallest element of the support, we can thus represent the function min with $H = \{(\mathcal{P}_3(\mathbb{Q}), 100, \mathbb{Q})\}$.

2.3 *Products*

The product $X \times Y$ of two nominal sets is again a nominal set and hence, it can be represented itself in terms of the dimension of each of its orbits as shown in Section 2.1. However, this approach has some disadvantages.

Example 11. We start by showing that the orbit structure of products can be non-trivial. Consider the product of $X = \mathbb{Q}$ and the set $Y = \{(a, b) \in \mathbb{Q}^2 \mid a < b\}$. This product consists of *five* orbits, more than one might naively expect from the fact that both sets are single-orbit:

$$\{(a, (b, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}, \quad \{(a, (a, b)) \mid a, b \in \mathbb{Q}, a < b\},$$
$$\{(b, (a, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}, \quad \{(b, (a, b)) \mid a, b \in \mathbb{Q}, a < b\},$$
$$\{(c, (a, b)) \mid a, b, c \in \mathbb{Q}, a < b < c\}.$$

We find that this product is represented by the multiset $\{2, 2, 3, 3, 3\}$. Unfortunately, this is not sufficient to accurately describe the product as it abstracts away from the relation between its elements with those in X and Y. In particular, it is not possible to reconstruct the projection maps from such a representation.

The essence of our representation of products is that each orbit O in the product $X \times Y$ is described entirely by the dimension of O together with the two (equivariant) projections $\pi_1 : O \to X$ and $\pi_2 : O \to Y$. This combination of the orbit and the two projection maps can already be represented using Propositions 6 and 10. However, as we will see, a combined representation for this has several advantages. For discussing such a representation, let us first introduce what it means for tuples of a set and two functions to be isomorphic:

Definition 12. Given nominal sets X, Y, Z_1 and Z_2, and equivariant functions $l_1 : Z_1 \to X$, $r_1 : Z_1 \to Y$, $l_2 : Z_2 \to X$ and $r_2 : Z_2 \to Y$, we define $(Z_1, l_1, r_1) \cong (Z_2, l_2, r_2)$ if there exists an isomorphism $h : Z_1 \to Z_2$ such that $l_1 = l_2 \circ h$ and $r_1 = r_2 \circ h$.

Our goal is to have a representation that, for each orbit O, produces a tuple (A, f_1, f_2) isomorphic to the tuple (O, π_1, π_2). The next lemma gives a characterisation that can be used to simplify such a representation.

Lemma 13. Let X and Y be nominal sets and $(x, y) \in X \times Y$. If C, C_x, and C_y are the least supports of (x, y), x, and y respectively, then $C = C_x \cup C_y$.

With Proposition 10 we represent the maps π_1 and π_2 by tuples (O, F_1, O_1) and (O, F_2, O_2) respectively. Using Lemma 13 and the definitions of F_1 and F_2, we see that at least one of $F_1(i)$ and $F_2(i)$ equals 1 for each i.

We can thus combine the strings F_1 and F_2 into a single string $P \in \{L, R, B\}^*$ as follows. We set $P(i) = L$ when only $F_1(i)$ is 1, $P(i) = R$ when only $F_2(i)$ is 1, and $P(i) = B$ when both are 1. The string P fully describes the strings F_1 and F_2. This process for constructing the string P gives it two useful properties. The number of Ls and Bs in the string gives the size dimension of O_1. Similarly, the number of Rs and Bs in the string gives the dimension of O_2. We will call strings with that property *valid*. In conclusion, to describe a single orbit of the product $X \times Y$, a valid string P together with the images of π_1 and π_2 is sufficient.

Definition 14. Let $P \in \{L, R, B\}^*$, and $O_1 \subseteq X$, $O_2 \subseteq Y$ be single-orbit sets. Given a tuple (P, O_1, O_2), where the string P is valid, define

$$[(P, O_1, O_2)]^t = (\mathcal{P}_{|P|}(\mathbb{Q}), f_{H_1}, f_{H_2}),$$

where $H_i = \{(\mathcal{P}_{|P|}(\mathbb{Q}), F_i, O_i)\}$ and the string F_1 is defined as the string P with Ls and Bs replaced by 1s and Rs by 0s. The string F_2 is similarly defined with the roles of L and R swapped.

Proposition 15. There exists a one-to-one correspondence between the orbits $O \subseteq X \times Y$, and tuples (P, O_1, O_2) satisfying $O_1 \subseteq X$, $O_2 \subseteq Y$, and where P is a valid string, such that $[(P, O_1, O_2)]^t \cong (O, \pi_1|_O, \pi_2|_O)$.

From the above proposition it follows that we can generate the product $X \times Y$ simply by enumerating all valid strings P for all pairs of orbits (O_1, O_2) of X and Y. Given this, we can calculate the multiset representation of a product from the multiset representations of both factors.

Theorem 16. For $X \cong [f]^o$ and $Y \cong [g]^o$ we have $X \times Y \cong [h]^o$, where

$$h(n) = \sum_{\substack{0 \leqslant i,j \leqslant n \\ i+j \geqslant n}} f(i)g(j)\binom{n}{j}\binom{j}{n-i}.$$

Example 17. To illustrate some aspects of the above representation, let us use it to calculate the product of Example 11. First, we observe that both $\mathbb{Q}$ and $S = \{(a, b) \in \mathbb{Q}^2 \mid a < b\}$ consist of a single orbit. Hence any orbit of the product corresponds to a triple $(P, \mathbb{Q}, S)$, where the string P satisfies $|P|_L + |P|_B = \dim(\mathbb{Q}) = 1$ and $|P|_R + |P|_B = \dim(S) = 2$. We can now find the orbits of the product $\mathbb{Q} \times S$ by enumerating all strings satisfying these equations. This yields

- LRR, corresponding to the orbit $\{(a, (b, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$,
- RLR, corresponding to the orbit $\{(b, (a, c)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$,
- RRL, corresponding to the orbit $\{(c, (a, b)) \mid a, b, c \in \mathbb{Q}, a < b < c\}$,
- RB, corresponding to the orbit $\{(b, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$, and
- BR, corresponding to the orbit $\{(a, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$.

Each product string fully describes the corresponding orbit. To illustrate, consider the string BR. The corresponding bit strings for the projection functions are $F_1 = 10$ and $F_2 = 11$. From the lengths of the string we conclude that the dimension of the orbit is 2. The string F_1 further tells us that the left element of the tuple consists only of the smallest element of the support. The string F_2 indicates that the right element of the tuple is constructed from both elements of the support. Combining this, we find that the orbit is $\{(a, (a, b)) \mid a, b \in \mathbb{Q}, a < b\}$.

2.4 *Summary*

We summarise our concrete representation in the following table. Propositions 6, 10 and 15 correspond to the three rows in the table.

Notice that in the case of maps and products, the orbits are inductively represented using the concrete representation. As a base case we can represent single orbits by their dimension.

Object	*Representation*
Single orbit O	Natural number $n = \dim(O)$
Nominal set $X = \cup_i O_i$	Multiset of these numbers
Map from single orbit $f\colon O \to Y$	The orbit $f(O)$ and a bit string F
Equivariant map $f\colon X \to Y$	Set of tuples $(O, F, f(O))$, one for each orbit
Orbit in a product $O \subseteq X \times Y$	The corresponding orbits of X and Y, and a string P relating their supports
Product $X \times Y$	Set of tuples (P, O_X, O_Y), one for each orbit

Table 6.1 Overview of representation.

3 *Implementation and Complexity of ONS*

The ideas outlined above have been implemented in a C++ library, ONS, and a Haskell library, ONS-HS.[20] We focus here on the C++ library only, as the Haskell one is very similar. The library can represent orbit-finite nominal sets and their products, (disjoint) unions, and maps. A full description of the possibilities is given in the documentation included with ONS.

As an example, the following program computes the product from Example 11. Initially, the program creates the nominal set A, containing the entirety of $\mathbb{Q}$. Then it creates a nominal set B, such that it consists of the orbit containing the element $(1, 2) \in \mathbb{Q} \times \mathbb{Q}$. For this, the library determines to which orbit of the product $\mathbb{Q} \times \mathbb{Q}$ the element $(1, 2)$ belongs, and then stores a description of the orbit as described in Section 2. Note that this means that it internally never needs to store the element used to create the orbit. The function `nomset_product` then uses the enumeration of product strings mentioned in Section 2.3 to calculate the product of A and B. Finally, it prints a representative element for each of the orbits in the product. These elements are constructed based on the description of the orbits stored, filled in to make their support equal to sets of the form $\{1, 2, ..., n\}$.

```
nomset<rational> A = nomset_rationals();
nomset<pair<rational, rational>> B({rational(1),rational(2)});
auto AtimesB = nomset_product(A, B);   // compute the product
for (auto orbit : AtimesB)
       cout << orbit.getElement() << " ";
```

Running this gives the following output (where `/1` signifies the denominator):

```
(1/1,(2/1,3/1)) (1/1,(1/1,2/1)) (2/1,(1/1,3/1))
(2/1,(1/1,2/1)) (3/1,(1/1,2/1))
```

[20] ONS can be found at https://github.com/davidv1992/ONS and ONS-HS can be found at https://github.com/Jaxan/ons-hs/.

Internally, `orbit` is implemented following the theory presented in Section 2, storing the dimension of the orbit it represents. It also contains sufficient information to reconstruct elements given their least support, such as the product string for orbits resulting from a product. The class `nomset` then uses a standard set data structure to store the collection of orbits contained in the nominal set it represents.

In a similar way, `eqimap` stores equivariant maps by associating each orbit in the domain with the image orbit and the string representing which of the least support to keep. This is stored using a map data structure. For both nominal sets and equivariant maps, the underlying data structure is currently implemented using trees.

3.1 *Complexity of operations*

Using the concrete representation of nominal sets, we can determine the complexity of common operations. To simplify such an analysis, we will make the following assumptions:

- The comparison of two orbits takes $O(1)$.
- Constructing an orbit from an element takes $O(1)$.
- Checking whether an element is in an orbit takes $O(1)$.

These assumptions are justified as each of these operations takes time proportional to the size of the representation of an individual orbit, which in practice is small and approximately constant. For instance, the orbit $\mathcal{P}_n(\mathbb{Q})$ is represented by just the integer n and its type.

Theorem 18. If nominal sets are implemented with a tree-based set structure (as in Ons), the complexity of the following set operations is as follows. Recall that $N(X)$ denotes the number of orbits of X. We use p and f to denote functions implemented in whatever way the user wants, which we assume to take $O(1)$ time. The software assumes these are equivariant, but this is not verified.

Operation	*Complexity*
Test $x \in X$	$O(\log N(X))$
Test $X \subseteq Y$	$O(\min(N(X) + N(Y), N(X) \log N(Y)))$
Calculate $X \cup Y$	$O(N(X) + N(Y))$
Calculate $X \cap Y$	$O(N(X) + N(Y))$
Calculate $\{x \in X \mid p(x)\}$	$O(N(X))$
Calculate $\{f(x) \mid x \in X\}$	$O(N(X) \log N(X))$
Calculate $X \times Y$	$O(N(X \times Y)) \subseteq O(3^{\dim(X)+\dim(Y)} N(X)N(Y))$

Table 6.2 Time complexity of operations on nominal sets.

Proof. Since most parts are proven similarly, we only include proofs for the first and last item.

Membership. To decide $x \in X$, we first construct the orbit containing x, which is done in constant time. Then we use a logarithmic lookup to decide whether this orbit is in our set data structure. Hence, membership checking is $O(\log(N(X)))$.

Products. Calculating the product of two nominal sets is the most complicated construction. For each pair of orbits in the original sets X and Y, all product strings need to be generated. Each product orbit itself is constructed in constant time. By generating these orbits in-order, the resulting set takes $O(N(X \times Y))$ time to construct.

We can also give an explicit upper bound for the number of orbits in terms of the input. Recall that orbits in a product are represented by strings of length at most $\dim(X) + \dim(Y)$. (If the string is shorter, we pad it with one of the symbols.) Since there are three symbols (L, R and B), the product of X and Y will have at most $3^{\dim(X)+\dim(Y)}N(X)N(Y)$ orbits. It follows that taking products has time complexity of $O(3^{\dim(X)+\dim(Y)}N(X)N(Y))$. □

4 *Results and evaluation in automata theory*

In this section we consider applications of nominal sets to automata theory. As mentioned in the introduction, nominal sets are used to formalise languages over infinite alphabets. These languages naturally arise as the semantics of register automata. The definition of register automata is not as simple as that of ordinary finite automata. Consequently, transferring results from automata theory to this setting often requires non-trivial proofs. Nominal automata, instead, are defined as ordinary automata by replacing finite sets with orbit-finite nominal sets. The theory of nominal automata is developed by Bojańczyk, et al. (2014) and it is shown that many algorithms, such as minimisation (based on the Myhill-Nerode equivalence), from automata theory transfer to nominal automata. Not all algorithms work: e.g., the subset construction fails for nominal automata.

As an example we consider the following language on rational numbers:

$$\mathcal{L}_{\text{int}} = \{a_1 b_1 \cdots a_n b_n \mid a_i, b_i \in \mathbb{Q}, a_i < a_{i+1} < b_{i+1} < b_i \text{ for all } i\}.$$

We call this language the *interval language* as a word $w \in \mathbb{Q}^*$ is in the language when it denotes a sequence of nested intervals. This language contains arbitrarily long words. For this language it is crucial to work with an infinite alphabet as for each finite set $C \subset \mathbb{Q}$, the restriction $\mathcal{L}_{\text{int}} \cap C^*$ is just a finite language. Note that the language is equivariant: $w \in \mathcal{L}_{\text{int}} \iff wg \in \mathcal{L}_{\text{int}}$ for any monotone bijection g, because nested intervals are preserved by monotone maps.[21] Indeed, $\mathcal{L}_{\text{int}}$ is a nominal set, although it is not orbit-finite.

Informally, the language $\mathcal{L}_{\text{int}}$ can be accepted by the automaton depicted in Figure 6.1. Here we allow the automaton to store rational numbers and compare them to

[21] The G-action on words is defined point-wise: $g(w_1 ... w_n) = (gw_1)...(gw_n)$.

new symbols. For example, the transition from q_2 to q_3 is taken if any value c between a and b is read and then the currently stored value a is replaced by c. For any other value read at state q_2 the automaton transitions to the sink state q_4. Such a transition structure is made precise by the notion of nominal automata.

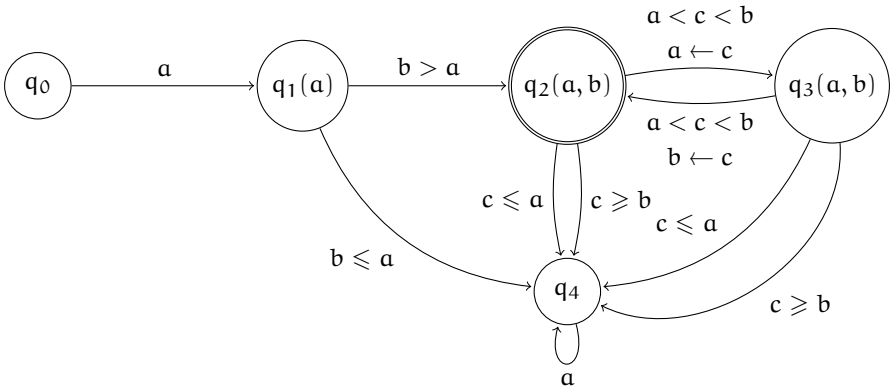


Figure 6.1 Example automaton that accepts the language $\mathcal{L}_{\text{int}}$.

Definition 19. A *nominal language* is an equivariant subset $L \subseteq A^*$ where A is an orbit-finite nominal set.

Definition 20. A *nominal deterministic finite automaton* is a tuple (S, A, F, δ), where S is an orbit-finite nominal set of states, A is an orbit-finite nominal set of symbols, $F \subseteq S$ is an equivariant subset of final states, and $\delta : S \times A \to S$ is the equivariant transition function.

Given a state $s \in S$, we define the usual acceptance condition: a word $w \in A^*$ is *accepted* if w denotes a path from s to a final state.

The automaton in Figure 6.1 can be formalised as a nominal deterministic finite automaton as follows. Let $S = \{q_0, q_4\} \cup \{q_1(a) \mid a \in \mathbb{Q}\} \cup \{q_2(a,b) \mid a < b \in \mathbb{Q}\} \cup \{q_3(a,b) \mid a < b \in \mathbb{Q}\}$ be the set of states, where the group action is defined as one would expect. The transition we described earlier can now be formally defined as $\delta(q_2(a,b), c) = q_3(c,b)$ for all $a < c < b \in \mathbb{Q}$. By defining δ on all states accordingly and defining the final states as $F = \{q_2(a,b) \mid a < b \in \mathbb{Q}\}$, we obtain a nominal deterministic automaton $(S, \mathbb{Q}, F, \delta)$. The state q_0 accepts the language $\mathcal{L}_{\text{int}}$.

We implement two algorithms on nominal automata, minimisation and learning, to benchmark Ons. The performance of Ons is compared to two existing libraries for computing with nominal sets, Nλ and Lois. The following automata will be used.

Random automata

As a primary test suite, we generate random automata as follows. The input alphabet is always $\mathbb{Q}$ and the number of orbits and dimension k of the state space S are fixed.

For each orbit in the set of states, its dimension is chosen uniformly at random between 0 and k, inclusive. Each orbit has a probability $\frac{1}{2}$ of consisting of accepting states.

To generate the transition function δ, we enumerate the orbits of $S \times \mathbb{Q}$ and choose a target state uniformly from the orbits S with small enough dimension. The bit string indicating which part of the support is preserved is then sampled uniformly from all valid strings. We will denote these automata as $\text{rand}_{N(S),k}$. The choices made here are arbitrary and only provide basic automata. We note that the automata are generated orbit-wise and this may favour our tool.

Structured automata

Besides random automata we wish to test the algorithms on more structured automata. We define the following automata.

FIFO(n) Automata accepting valid traces of a finite FIFO data structure of size n. The alphabet is defined by two orbits: $\{\texttt{Put}(a) \mid a \in \mathbb{Q}\}$ and $\{\texttt{Get}(a) \mid a \in \mathbb{Q}\}$.

$ww(n)$ Automata accepting the language of words of the form ww, where $w \in \mathbb{Q}^n$.

$\mathcal{L}_{\max}$ The language $\mathcal{L}_{\max} = \{wa \in \mathbb{Q}^* \mid a = \max(w_1, ..., w_n)\}$ where the last symbol is the maximum of previous symbols.

$\mathcal{L}_{\text{int}}$ The language accepting a series of nested intervals, as defined before.

In Table 6.3 we report the number of orbits for each automaton. The first two classes of automata are described in Chapter 5. These two classes are also equivariant w.r.t. the equality symmetry.

Extra structure allows the automata to be encoded more efficiently, as we do not need to encode a transition for each orbit in $S \times A$. Instead, a more symbolic encoding is possible. Both LOIS and Nλ allow to use this more symbolic representation. Our tool, ONS, only works with nominal sets and the input data needs to be provided orbit-wise. Where applicable, the automata listed above were generated using the code from Moerman, et al. (2017), ported to the other libraries as needed.

4.1 Minimising nominal automata

For languages recognised by nominal DFAs, a Myhill-Nerode theorem holds which relates states to right congruence classes. This guarantees the existence of unique minimal automata. We say an automaton is *minimal* if its set of states has the least number of orbits and each orbit has the smallest dimension possible.[22] We generalise

[22] Abstractly, an automaton is minimal if it has no proper quotients. Minimal deterministic automata are unique up to isomorphism.

Moore's minimisation algorithm to nominal DFAs (Algorithm 6.1) and analyse its time complexity using the bounds from Section 3.

Require: Nominal automaton $M = (S, A, F, \delta)$
Ensure: Minimal nominal automaton equivalent to M
1 $i \leftarrow 0$
2 $\equiv_{-1} \leftarrow S \times S$
3 $\equiv_0 \leftarrow F \times F \cup (S \backslash F) \times (S \backslash F)$
4 **while** $\equiv_i \neq \equiv_{i-1}$ **do**
5 $\quad \equiv_{i+1} \leftarrow \{(q_1, q_2) \mid (q_1, q_2) \in \equiv_i \wedge \forall a \in A, (\delta(q_1, a), \delta(q_2, a)) \in \equiv_i\}$
6 $\quad i \leftarrow i + 1$
7 **end while**
8 $E \leftarrow S/_{\equiv_i}$
9 $F_E \leftarrow \{e \in E \mid \forall s \in e, s \in F\}$
10 Let δ_E be the map such that, if $s \in e$ and $\delta(s, a) \in e'$, then $\delta_E(e, a) = e'$
11 **return** (E, A, F_E, δ_E)

Algorithm 6.1 Moore's minimisation algorithm for nominal DFAs.

Theorem 21. The runtime complexity of Moore's algorithm on nominal deterministic automata is $O(3^{5k} k N(S)^3 N(A))$, where $k = \dim(S \cup A)$.

Proof. This is shown by counting operations, using the complexity results of set operations stated in Theorem 18. We first focus on the while loop on lines 4–7. The runtime of an iteration of the loop is determined by line 5, as this is the most expensive step. Since the dimensions of S and A are at most k, computing $S \times S \times A$ takes $O(N(S)^2 N(A) 3^{5k})$. Filtering $S \times S$ using that then takes $O(N(S)^2 3^{2k})$. The time to compute $S \times S \times A$ dominates, hence each iteration of the loop takes $O(N(S)^2 N(A) 3^{5k})$.

Next, we need to count the number of iterations of the loop. Each iteration of the loop gives rise to a new partition, refining the previous partition. Furthermore, every partition generated is equivariant. Note that this implies that each refinement of the partition does at least one of two things: distinguish between two orbits of S previously in the same element(s) of the partition, or distinguish between two members of the same orbit previously in the same element of the partition. The former can happen only $N(S) - 1$ times, as after that there are no more orbits lumped together. The latter can only happen $\dim(S)$ times per orbit, because each such a distinction between elements is based on splitting on the value of one of the elements of the support. Hence, after $\dim(S)$ times on a single orbit, all elements of the support are used up. Combining this, the longest chain of partitions of S has length at most $O(kN(S))$.

Since each partition generated in the loop is unique, the loop cannot run for more iterations than the length of the longest chain of partitions on S. It follows that

there are at most $O(kN(S))$ iterations of the loop, giving the loop a complexity of $O(kN(S)^3N(A)3^{5k})$

The remaining operations outside the loop have a lower complexity than that of the loop, hence the complexity of Moore's minimisation algorithm for a nominal automaton is $O(kN(S)^3N(A)3^{5k})$. □

The above theorem shows in particular that minimisation of nominal automata is fixed-parameter tractable (FPT) with the dimension as fixed parameter. The complexity of Algorithm 6.1 for nominal automata is very similar to the $O((\#S)^3 \# A)$ bound given by a naive implementation of Moore's algorithm for ordinary DFAs. This suggest that it is possible to further optimise an implementation with similar techniques used for ordinary automata.

Implementations

We implemented the minimisation algorithm in Ons. For Nλ and Lois we used their implementations of Moore's minimisation algorithm (Klin & Szynwelski, 2016 and Kopczyński & Toruńczyk, 2016 and 2017). For each of the libraries, we wrote routines to read in an automaton from a file and, for the structured test cases, to generate the requested automaton. For Ons, all automata were read from file. The output of these programs was manually checked to see if the minimisation was performed correctly.

Results

The results (shown in Table 6.3) for random automata show a clear advantage for Ons, which is capable of running all supplied testcases in less than one second. This in contrast to both Lois and Nλ, which take more than 2 hours on the largest random automata.

The results for structured automata show a clear effect of the extra structure. Both Nλ and Lois remain capable of minimising the automata in reasonable amounts of time for larger sizes. In contrast, Ons benefits little from the extra structure. Despite this, it remains viable: even for the larger cases it falls behind significantly only for the largest FIFO automaton and the two largest *ww* automata.

4.2 Learning nominal automata

Another application that we implemented in Ons is *automata learning*. The aim of automata learning is to infer an unknown regular language $\mathcal{L}$. We use the framework of active learning as set up by Angluin (1987) where a learning algorithm can query an oracle to gather information about $\mathcal{L}$. Formally, the oracle can answer two types of queries:

Type	$N(S)$	$N(S^{min})$	Ons (s)	Gen (s)	Nλ (s)	Lois (s)
$rand_{5,1}$ (x10)	5	n/a	0.02	n/a	0.82	3.14
$rand_{10,1}$ (x10)	10	n/a	0.03	n/a	17.03	92
$rand_{10,2}$ (x10)	10	n/a	0.09	n/a	2114	∞
$rand_{15,1}$ (x10)	15	n/a	0.04	n/a	87	620
$rand_{15,2}$ (x10)	15	n/a	0.11	n/a	3346	∞
$rand_{15,3}$ (x10)	15	n/a	0.46	n/a	∞	∞
FIFO(2)	13	6	0.01	0.01	1.37	0.24
FIFO(3)	65	19	0.38	0.09	11.59	2.4
FIFO(4)	440	94	39.11	1.60	76	14.95
FIFO(5)	3686	635	∞	39.78	402	71
$ww(2)$	8	8	0.00	0.00	0.14	0.03
$ww(3)$	24	24	0.19	0.02	0.88	0.16
$ww(4)$	112	112	26.44	0.25	3.41	0.61
$ww(5)$	728	728	∞	6.37	10.54	1.80
$\mathcal{L}_{max}$	5	3	0.00	0.00	2.06	0.03
$\mathcal{L}_{int}$	5	5	0.00	0.00	1.55	0.03

Table 6.3 Running times for Algorithm 6.1 implemented in the three libraries. $N(S)$ is the size of the input and $N(S^{min})$ the size of the minimal automaton. For Ons, the time used to generate the automaton is reported separately (in grey). Timeouts are indicated by ∞.

– *membership queries*, where a query consists of a word $w \in A^*$ and the oracle replies whether $w \in \mathcal{L}$, and
– *equivalence queries*, where a query consists of an automaton $\mathcal{H}$ and the oracle replies positively if $\mathcal{L}(\mathcal{H}) = \mathcal{L}$ or provides a counterexample if $\mathcal{L}(\mathcal{H}) \neq \mathcal{L}$.

With these queries, the L* algorithm can learn regular languages efficiently (Angluin, 1987). In particular, it learns the unique minimal automaton for $\mathcal{L}$ using only finitely many queries. The L* algorithm has been generalised to νL* in order to learn *nominal* regular languages. In particular, it learns a nominal DFA (over an infinite alphabet) using only finitely many queries. We implement νL* in the presented library and compare it to its previous implementation in Nλ. The algorithm is not polynomial, unlike the minimisation algorithm described above. However, the authors conjecture that there is a polynomial algorithm.[23] For the correctness, termination, and comparison with other learning algorithms see Chapter 5.

[23] See https://joshuamoerman.nl/papers/2017/17popl-learning-nominal-automata.html for a sketch of the polynomial algorithm.

Implementations

Both implementations in Nλ and Ons are direct implementations of the pseudocode for νL* with no further optimisations. The authors of Lois implemented νL* in their library as well.[24] They reported similar performance as the implementation in Nλ (private communication). Hence we focus our comparison on Nλ and Ons. We use the variant of νL* where counterexamples are added as columns instead of prefixes.

The implementation in Nλ has the benefit that it can work with different symmetries. Indeed, the structured examples, FIFO and *ww*, are equivariant w.r.t. the equality symmetry as well as the total order symmetry. For that reason, we run the Nλ implementation using both the equality symmetry and the total order symmetry on those languages. For the languages $\mathcal{L}_{\max}$, $\mathcal{L}_{\text{int}}$ and the random automata, we can only use the total order symmetry.

To run the νL* algorithm, we implement an external oracle for the membership queries. This is akin to the application of learning black box systems (see Vaandrager, 2017). For equivalence queries, we constructed counterexamples by hand. All implementations receive the same counterexamples. We measure CPU time instead of real time, so that we do not account for the external oracle.

Results

The results (Table 6.4) for random automata show an advantage for Ons. Additionally, we report the number of membership queries, which can vary for each implementation as some steps in the algorithm depend on the internal ordering of set data structures.

In contrast to the case of minimisation, the results suggest that Nλ cannot exploit the logical structure of FIFO(n), $\mathcal{L}_{\max}$ and $\mathcal{L}_{\text{int}}$ as it is not provided a priori. For $ww(2)$ we inspected the output on Nλ and saw that it learned some logical structure (e.g., it outputs $\{(a, b) \mid a \neq b\}$ as a single object instead of two orbits $\{(a, b) \mid a < b\}$ and $\{(a, b) \mid b < a\}$). This may explain why Nλ is still competitive. For languages which are equivariant for the equality symmetry, the Nλ implementation using the equality symmetry can learn with much fewer queries. This is expected as the automata themselves have fewer orbits. It is interesting to see that these languages can be learned more efficiently by choosing the right symmetry.

5 *Related work*

As stated in the introduction, Nλ by Klin and Szynwelski (2016) and Lois by Kopczyński and Toruńczyk (2016) use first-order formulas to represent nominal sets and use SMT solvers to manipulate them. This makes both libraries very flexible and they indeed

[24] Can be found on https://github.com/eryxcc/lois/blob/master/tests/learning.cpp.

			Ons		Nλ^{ord}		Nλ^{eq}
Model	N(S)	dim(S)	time (s)	MQs	time (s)	MQs	time (s)	MQs
$rand_{5,1}$	4	1	127	2321	2391	1243		
$rand_{5,1}$	5	1	0.12	404	2434	435		
$rand_{5,1}$	3	0	0.86	499	1819	422		
$rand_{5,1}$	5	1	∞	n/a	∞	n/a		
$rand_{5,1}$	4	1	0.08	387	2097	387		
FIFO(1)	3	1	0.04	119	3.17	119	1.76	51
FIFO(2)	6	2	1.73	2655	392	3818	40.00	434
FIFO(3)	19	3	2794	298400	∞	n/a	2047	8151
ww(1)	4	1	0.42	134	2.49	77	1.47	30
ww(2)	8	2	266	3671	228	2140	30.58	237
ww(3)	24	3	∞	n/a	∞	n/a	∞	n/a
$\mathcal{L}_{max}$	3	1	0.01	54	3.58	54		
$\mathcal{L}_{int}$	5	2	0.59	478	83	478		

Table 6.4 Running times and number of membership queries for the νL* algorithm. For Nλ we used two version: Nλ^{ord} uses the total order symmetry Nλ^{eq} uses the equality symmetry. Timeouts are indicated by ∞.

implement the equality symmetry as well as the total order symmetry. As their representation is not unique, the efficiency depends on how the logical formulas are constructed. As such, they do not provide complexity results. In contrast, our direct representation allows for complexity results (Section 3) and leads to different performance characteristics (Section 4).

A second big difference is that both Nλ and Lois implement a "programming paradigm" instead of just a library. This means that they overload natural programming constructs in their host languages (Haskell and C++ respectively). For programmers this means they can think of infinite sets without having to know about nominal sets.

It is worth mentioning that an older (unreleased) version of Nλ implemented nominal sets with orbits instead of SMT solvers (Bojańczyk, et al., 2012). However, instead of characterising orbits (e.g., by its dimension), they represent orbits by a representative element. Klin and Szynwelski (2016) reported that the current version is faster.

The theoretical foundation of our work is the main representation theorem by Bojańczyk, et al. (2014). We improve on that by instantiating it to the total order symmetry and distil a concrete representation of nominal sets. As far as we know, we provide the first implementation of their representation theory.

Another tool using nominal sets is Mihda by Ferrari, et al. (2005). Here, only the equality symmetry is implemented. This tool implements a translation from π-calculus to history-dependent automata (HD-automata) with the aim of minimisation and checking bisimilarity. The implementation in OCaml is based on *named sets*, which are finite representations for nominal sets. The theory of named sets is well-studied

and has been used to model various behavioural models with local names. For those results, the categorical equivalences between named sets, nominal sets and a certain (pre)sheaf category have been exploited (Ciancia, et al., 2010 and Ciancia & Montanari, 2010). The total order symmetry is not mentioned in their work. We do, however, believe that similar equivalences between categories can be stated. Interestingly, the product of named sets is similar to our representation of products of nominal sets: pairs of elements together with data which denotes the relation between data values.

Fresh OCaml by Shinwell and Pitts (2005) and Nominal Isabelle by Urban and Tasson (2005) are both specialised in name-binding and α-conversion used in proof systems. They only use the equality symmetry and do not provide a library for manipulating nominal sets. Hence they are not suited for our applications.

On the theoretical side, there are many complexity results for register automata (Grigore & Tzevelekos, 2016 and Murawski, et al., 2015). In particular, we note that problems such as emptiness and equivalence are NP-hard depending on the type of register automaton. Recently, Murawski, et al. (2018) showed that equivalence of unique-valued deterministic register automata can be decided in polynomial time. These results do not easily compare to our complexity results for minimisation. One difference is that we use the total order symmetry, where the local symmetries are always trivial (Lemma 3). As a consequence, all the complexity required to deal with groups vanishes. Rather, the complexity is transferred to the input of our algorithms, because automata over the equality symmetry require more orbits when expressed over the total order symmetry. Another difference is that register automata allow for duplicate values in the registers. In nominal automata, such configurations will be encoded in different orbits.

Orthogonal to nominal automata, there is the notion of symbolic automata (D'Antoni & Veanes, 2017 and Maler & Mens, 2017). These automata are also defined over infinite alphabets but they use predicates on transitions, instead of relying on symmetries. Symbolic automata are finite state (as opposed to infinite state nominal automata) and do not allow for storing values. However, they do allow for general predicates over an infinite alphabet, including comparison to constants.

6 *Conclusion and Future Work*

We presented a concrete finite representation for nominal sets over the total order symmetry. This allowed us to implement a library, Ons, and provide complexity bounds for common operations. The experimental comparison of Ons against existing solutions for automata minimisation and learning show that our implementation is much faster in many instances. As such, we believe Ons is a promising implementation of nominal techniques.

A natural direction for future work is to consider other symmetries, such as the equality symmetry. Here, we may take inspiration from existing tools such as Mihda

(see Section 5). Another interesting question is whether it is possible to translate a nominal automaton over the total order symmetry which accepts an equality language to an automaton over the equality symmetry. This would allow one to efficiently move between symmetries. Finally, our techniques can potentially be applied to timed automata by exploiting the intriguing connection between the nominal automata that we consider and timed automata (Bojańczyk & Lasota, 2012).

Acknowledgement

We would like to thank Szymon Toruńczyk and Eryk Kopczyński for their prompt help when using the Lois library. For general comments and suggestions we would like to thank Ugo Montanari and Niels van der Weide. At last, we want to thank the anonymous reviewers for their comments.

Chapter 7
Separation and Renaming in Nominal Sets

Joshua Moerman
Radboud University

Jurriaan Rot
Radboud University

Abstract

Nominal sets provide a foundation for reasoning about names. They are used primarily in syntax with binders, but also, e.g., to model automata over infinite alphabets. In this chapter, nominal sets are related to *nominal renaming sets*, which involve arbitrary substitutions rather than permutations, through a categorical adjunction. In particular, the separated product of nominal sets is related to the Cartesian product of nominal renaming sets. Based on these results, we define the new notion of *separated nominal automata*. These efficiently recognise nominal languages, provided these languages are renaming sets. In such cases, moving from the existing notion of nominal automata to separated automata can lead to an exponential reduction of the state space.

This chapter is based on the following submission:
Moerman, J. & Rot, J. (2019). *Separation and Renaming in Nominal Sets*. (Under submission)

Nominal sets are abstract sets which allow one to reason over sets with names, in terms of permutations and symmetries. Since their introduction in computer science by Gabbay and Pitts (1999), they have been widely used for implementing and reasoning over syntax with binders (see the book of Pitts, 2013). Further, nominal techniques have been related to computability theory (Bojańczyk, et al., 2013) and automata theory (Bojańczyk, et al., 2014), where they provide an elegant means of studying languages over infinite alphabets. This embeds nominal techniques in a broader setting of *symmetry aware computation* (Pitts, 2016).

Gabbay, one of the pioneers of nominal techniques described a variation on the theme: *nominal renaming sets* (Gabbay, 2007 and Gabbay & Hofmann, 2008). Nominal renaming sets are equipped with a monoid action of arbitrary (possibly non-injective) substitution of names, in contrast to nominal sets, which only involve a group action of permutations.

In this paper, the motivation for using nominal renaming sets comes from automata theory over infinite alphabets. Certain languages form nominal renaming sets, which means that they are closed under all possible substitutions on atoms. In order to obtain efficient automata-theoretic representations of such languages, we systematically relate nominal renaming sets to nominal sets.

We start by establishing a categorical adjunction in Section 2:

$$\mathsf{Pm\text{-}Nom} \underset{\mathsf{U}}{\overset{\mathsf{F}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathsf{Sb\text{-}Nom}$$

where Pm-Nom is the usual category of nominal sets and Sb-Nom the category of nominal renaming sets. The right adjoint U simply forgets the action of non-injective substitutions. The left adjoint F freely extends a nominal set with elements representing the application of such substitutions. For instance, F maps the nominal set $\mathbb{A}^{(*)}$ of all words consisting of distinct atoms to the nominal renaming set $\mathbb{A}^*$ consisting of all words over the atoms.

In fact, the latter follows from one of the main results of this paper: F maps the *separated product* $X * Y$ of nominal sets to the Cartesian product of nominal renaming sets. Additionally, under certain conditions, U maps the exponent to the *magic wand* $X \mathrel{-\!\!*} Y$, which is the right adjoint of the separated product. The separated product consists of those pairs whose elements have disjoint supports. This is relevant for name abstraction (Pitts, 2013), and has also been studied in the setting of presheaf categories, aimed towards separation logic (O'Hearn, 2003).

We apply these connections between nominal sets and renaming sets in the context of automata theory. Nominal automata are an elegant model for recognising languages over infinite alphabets. They are expressively equivalent to the more classical register automata (Bojańczyk, 2018, Theorem 6.5), and have appealing properties that register automata lack, such as unique minimal automata. However, moving from register

automata to nominal automata can lead to an exponential blow-up in the number of states.[25]

As a motivating example, we consider a language modelling an n-bounded FIFO queue. The input alphabet is given by $\Sigma = \{\mathtt{Put}(a) \mid a \in \mathbb{A}\} \cup \{\mathtt{Pop}\}$, and the output alphabet by $O = \mathbb{A} \cup \{\bot\}$ (here $\bot$ is a *null* value). The language $L_n : \Sigma^* \to O$ maps a sequence of queue operations to the resulting top element when starting from the empty queue, or to $\bot$ if this is undefined. The language L_n can be recognised by a nominal automaton, but this requires an exponential number of states in n, as the automaton distinguishes internally between all possible equalities among elements in the queue.

Based on the observation that L_n is a nominal renaming set, we can come up with a *linear* automata-theoretic representation. To this end, we define the new notion of *separated nominal automaton*, where the transition function is only defined for pairs of states and letters with a disjoint support (Section 3). Using the aforementioned categorical framework, we find that such separated automata recognise languages which are nominal renaming sets. Although separated nominal automata are not as expressive as classical nominal automata, they can be much smaller. In particular, in the FIFO example, the reachable part of the separated automaton obtained from the original nominal automaton has $n + 1$ states, thus dramatically reducing the number of states. We expect that such a reduction is useful in many applications, such as automata learning (Chapter 5).

In summary, the main contributions of this paper are the adjunction between nominal sets and nominal renaming sets, the relation between separated product and the Cartesian product of renaming sets, and the application to automata theory. We conclude with a coalgebraic account of separated automata in Section 3.1. In particular, we justify the semantics of separated automata by showing how it arises through a final coalgebra, obtained by lifting the adjunction to categories of coalgebras. The last section is orthogonal to the other results, and background knowledge of coalgebra is needed only there.

1 Monoid actions and nominal sets

In order to capture both the standard notion of nominal sets by Pitts (2013) and sets with more general renaming actions by Gabbay and Hofmann (2008), we start by defining monoid actions.

Definition 1. Let $(M, \cdot, 1)$ be a monoid. An M-*set* is a set X together with a function $\cdot : M \times X \to X$ such that $1 \cdot x = x$ and $m \cdot (n \cdot x) = (m \cdot n) \cdot x$ for all $m, n \in M$ and $x \in X$. The function $\cdot$ is called an M-*action* and $m \cdot x$ is often written by juxtaposition mx. A function $f : X \to Y$ between two M-sets is M-*equivariant* if $m \cdot f(x) = f(m \cdot x)$ for

[25] Here 'number of states' refers to the number of orbits in the state space.

all $m \in M$ and $x \in X$. The class of M-sets together with equivariant maps forms a category M-Set.

Let $\mathbb{A} = \{a, b, c, ...\}$ be a countable infinite set of *atoms*. The two main instances of M considered in this paper are the monoid

$$\mathsf{Sb} = \{m \colon \mathbb{A} \to \mathbb{A} \mid m(a) \neq a \text{ for finitely many } a\}$$

of all (finite) substitutions (with composition as multiplication), and the monoid

$$\mathsf{Pm} = \{g \in \mathsf{Sb} \mid g \text{ is a bijection}\}$$

of all (finite) permutations. Since Pm is a submonoid of Sb, any Sb-set is also a Pm-set; and any Sb-equivariant map is also Pm-equivariant. This gives rise to a forgetful functor

$$U \colon \mathsf{Sb\text{-}Set} \to \mathsf{Pm\text{-}Set}.$$

The set $\mathbb{A}$ is an Sb-set by defining $m \cdot a = m(a)$. Given an M-set X, the set $\mathcal{P}(X)$ of subsets of X is an M-set, with the action defined by direct image.

For a Pm-set X, the *orbit* of an element x is the set $\mathrm{orb}(x) = \{g \cdot x \mid g \in \mathsf{Pm}\}$. We say X is *orbit-finite* if the set $\{\mathrm{orb}(s) \mid x \in X\}$ is finite.

For any monoid M, the category M-Set is symmetric monoidal closed. The product of two M-sets is given by the Cartesian product, with the action defined pointwise: $m \cdot (x, y) = (m \cdot x, m \cdot y)$. In M-Set, the exponent $X \to^M Y$ is given by the set $\{f \colon M \times X \to Y \mid f \text{ is equivariant}\}$.[26] The action on such an $f \colon M \times X \to Y$ is defined by $(m \cdot f)(n, x) = f(mn, x)$. A good introduction to the construction of the exponent is given by Simmons (n.d.). If M is a group, a simpler description of the exponent may be given, carried by the set of all functions $f \colon X \to Y$, with the action given by $(g \cdot f)(x) = g \cdot f(g^{-1} \cdot x)$.

1.1 *Nominal* M*-sets*

The notion of *nominal* set is usually defined w.r.t. a Pm-action. Here, we use the generalisation to Sb-actions from Gabbay and Hofmann (2008). Throughout this section, let M denote a submonoid of Sb.

Definition 2. Let X be an M-set, and $x \in X$ an element. A set $C \subset \mathbb{A}$ is an (M-)*support* of x if for all $m_1, m_2 \in M$ s.t. $m_1|_C = m_2|_C$ we have $m_1 x = m_2 x$. An M-set X is called *nominal* if every element x has a finite M-support.

Nominal M-sets and equivariant maps form a full subcategory of M-Set, denoted by M-Nom. The M-set $\mathbb{A}$ of atoms is nominal. The powerset $\mathcal{P}(X)$ of a nominal set is not nominal in general; the restriction to finitely supported elements is.

[26] If we write a regular arrow $\to$, then we mean a map in the category. Exponent objects will always be denoted by annotated arrows.

If M is a group, then the notion of support can be simplified by using inverses. To see this, first note that, given elements $g_1, g_2 \in M$, $g_1|_C = g_2|_C$ can equivalently be written as $g_1 g_2^{-1}|_C = \mathrm{id}|_C$. Second, the statement $xg_1 = xg_2$ can be expressed as $xg_1 g_2^{-1} = x$. Hence, C is a support iff $g|_C = \mathrm{id}_C$ implies $gx = x$ for all g, which is the standard definition for nominal sets over a group (Pitts, 2013). Surprisingly, Gabbay and Hofmann (2008) show a similar characterisation also holds for Sb-sets. Moreover, recall that every Sb-set is also a Pm-set; the associated notions of support coincide on nominal Sb-sets, as shown by the following result. In particular, this means that the forgetful functor restricts to $U\colon \mathsf{Sb\text{-}Nom} \to \mathsf{Pm\text{-}Nom}$.

Lemma 3. (Theorem 4.8 from Gabbay, 2007) Let X be a nominal Sb-set, $x \in X$, and $C \subset \mathbb{A}$. Then C is an Sb-support of x iff it is a Pm-support of x.

Remark 4. It is not true that any Pm-support is an Sb-support. The condition that X is nominal, in the above lemma, is crucial. Let $X = \mathbb{A} + 1$ and define the following Sb-action: $m \cdot a = m(a)$ if m is injective, $m \cdot a = *$ if m is non-injective, and $m \cdot * = *$. This is a well-defined Sb-set, but is *not nominal*. Now consider $U(X)$, this is the Pm-set $\mathbb{A} + 1$ with the natural action, which is a *nominal* Pm-set! In particular, as a Pm-set each element has a finite support, but as a Sb-set the supports are infinite.

This counterexample is similar to the "exploding nominal sets" of Gabbay (2007), but even worse behaved. We like to call them *nuclear sets*, since an element will collapse when hit by a non-injective map, no matter how far away the non-injectivity occurs.

For $M \in \{\mathsf{Sb}, \mathsf{Pm}\}$, any element $x \in X$ of a nominal M-set X has a least finite support (w.r.t. set inclusion). We denote the least finite support of an element $x \in X$ by $\mathrm{supp}(x)$. Note that by Lemma 3, the set $\mathrm{supp}(x)$ is independent of whether a nominal Sb-set X is viewed as an Sb-set or a Pm-set. The *dimension* of X is given by $\dim(X) = \max\{|\mathrm{supp}(x)| \mid x \in X\}$, where $|\mathrm{supp}(x)|$ is the cardinality of $\mathrm{supp}(x)$.

We list some basic properties of nominal M-sets, which have known counterparts for the case that M is a group (Bojańczyk, et al., 2014), and when $M = \mathsf{Sb}$ (Gabbay & Hofmann, 2008).

Lemma 5. Let X be an M-nominal set. If C supports an element $x \in X$, then $m \cdot C$ supports $m \cdot x$ for all $m \in M$. Moreover, any $g \in \mathsf{Pm}$ preserves least supports: $g \cdot \mathrm{supp}(x) = \mathrm{supp}(gx)$.

The latter equality does not hold in general for a monoid M. For instance, the exploding nominal renaming sets by Gabbay and Hofmann (2008) give counterexamples for $M = \mathsf{Sb}$.

Lemma 6. Given M-nominal sets X, Y and a map $f\colon X \to Y$, if f is M-equivariant and C supports an element $x \in X$, then C supports $f(x)$.

The category M-Nom is symmetric monoidal closed, with the product inherited from M-Set, thus simply given by Cartesian product. The exponent is given by the restriction of the exponent $X \to^M Y$ in M-Set to the set of finitely supported functions, denoted by $X \to^M_{\mathrm{fs}} Y$. This is similar to the exponents of nominal sets with 01-substitutions from Pitts (2014).

Remark 7. Gabbay and Hofmann (2008) give a different presentation of the exponent in M-Nom, based on a certain extension of partial functions. We prefer the previous characterisation, as it is derived in a straightforward way from the exponent in M-Set.

1.2 *Separated product*

Definition 8. Two elements $x, y \in X$ of a Pm-nominal set are called *separated*, denoted by $x \# y$, if there are disjoint sets $C_1, C_2 \subset \mathbb{A}$ such that C_1 supports x and C_2 supports y. The *separated product* of Pm-nominal sets X and Y is defined as

$$X * Y = \{(x, y) \mid x \# y\}.$$

We extend the separated product to the *separated power*, defined by $X^{(0)} = 1$ and $X^{(n+1)} = X^{(n)} * X$, and the *set of separated words* $X^{(*)} = \cup_i X^{(i)}$. The separated product is an equivariant subset $X * Y \subseteq X \times Y$. Consequently, we have equivariant projection maps $X * Y \to X$ and $X * Y \to Y$.

Example 9. Two finite sets $C, D \subset \mathbb{A}$ are separated precisely when they are disjoint. An important example is the set $\mathbb{A}^{(*)}$ of separated words over the atoms: it consists of those words where all letters are distinct.

The separated product gives rise to another symmetric closed monoidal structure on Pm-Nom, with 1 as unit, and the exponential object given by *magic wand* $X \mathrel{-\!\!*} Y$. An explicit characterisation of $X \mathrel{-\!\!*} Y$ is not needed in the remainder of this chapter, but for a complete presentation we briefly recall the description from Schöpp (2006) (see also the book of Pitts, 2013 and the paper of Clouston, 2013). First, define a Pm-action on the set of partial functions $f : X \rightharpoonup Y$ by $(g \cdot f)(x) = g \cdot f(g^{-1} \cdot x)$ if $f(g^{-1} \cdot x)$ is defined. Now, such a partial function $f : X \rightharpoonup Y$ is called *separating* if f is finitely supported, $f(x)$ is defined iff $f \# x$, and $\mathrm{supp}(f) = \cup_{x \in \mathrm{dom}(f)} \mathrm{supp}(f(x)) \setminus \mathrm{supp}(x)$. Finally, $X \mathrel{-\!\!*} Y = \{f : X \rightharpoonup Y \mid f \text{ is separating}\}$. We refer to the thesis of Schöpp (2006) (Section 3.3.1) for a proof and explanation.

Remark 10. The special case $\mathbb{A} \mathrel{-\!\!*} Y$ coincides with $[\mathbb{A}]Y$, the set of *name abstractions* (Pitts, 2013). The latter is generalised to $[X]Y$ by Clouston (2013), but it is shown there that the coincidence $[X]Y \cong (X \mathrel{-\!\!*} Y)$ only holds under strong assumptions (including that X is single-orbit).

Remark 11. An analogue of the separated product does not seem to exist for nominal Sb-sets. For instance, consider the set $\mathbb{A} \times \mathbb{A}$. As a Pm-set, it has four equivariant subsets: $\emptyset, \Delta(\mathbb{A}) = \{(a, a) \mid a \in \mathbb{A}\}$, $\mathbb{A} * \mathbb{A}$, and $\mathbb{A} \times \mathbb{A}$. However, the set $\mathbb{A} * \mathbb{A}$ is not an equivariant subset when considering $\mathbb{A} \times \mathbb{A}$ as an Sb-set.

2 *A monoidal construction from Pm-sets to Sb-sets*

In this section, we provide a free construction, extending nominal Pm-sets to nominal Sb-sets. We use this as a basis to relate the separated product and exponent (in Pm-Nom) to the product and exponent in Sb-Nom. The main results are:

- the forgetful functor $U : \mathsf{Sb\text{-}Nom} \to \mathsf{Pm\text{-}Nom}$ has a left adjoint F (Theorem 16);
- this F is monoidal: it maps separated products to products (Theorem 17);
- U maps the exponent object in Sb-Nom to the right adjoint $-\!\!*$ of the separated product, if the domain has dimension $\leqslant 1$ (Theorem 24, Corollary 25).

Together, these results form the categorical infrastructure to relate nominal languages to separated languages and automata in Section 3.

Definition 12. Given a Pm-nominal set X, we define a nominal Sb-set $F(X)$ as follows. Define the set

$$F(X) = \{(m, x) \mid m \in \mathsf{Sb}, x \in X\}/_\sim,$$

where $\sim$ is the least equivalence relation containing:

$$(m, gx) \sim (mg, x),$$
$$(m, x) \sim (m', x) \quad \text{if } m|_C = m'|_C \text{ for a Pm-support } C \text{ of } x,$$

for all $x \in X$, $m, m' \in \mathsf{Sb}$ and $g \in \mathsf{Pm}$. The equivalence class of a pair (m, x) is denoted by $[m, x]$. We define an Sb-action on $F(X)$ as $n \cdot [m, x] = [nm, x]$.

Well-definedness is proved as part of Proposition 15 below. Informally, an equivalence class $[m, x] \in F(X)$ behaves "as if m acted on x". The first equation of $\sim$ ensures compatibility with the Pm-action on x, and the second equation ensures that $[m, x]$ only depends the relevant part of m. The following characterisation of $\sim$ is useful in proofs.

Lemma 13. We have $(m_1, x_1) \sim (m_2, x_2)$ iff there is a permutation $g \in \mathsf{Pm}$ such that $gx_1 = x_2$ and $m_1|_C = m_2 g|_C$, for C some Pm-support of x_1.

Remark 14. The first relation of $\sim$ in Definition 12 comes from the construction of "extension of scalars" in commutative algebra (see Atiyah & MacDonald, 1969). In that context, one has a ring homomorphism $f : A \to B$ and an A-module M and wishes

to obtain a B-module. This is constructed by the tensor product $B \otimes_A M$ and it is here that the relation $(b, am) \sim (ba, m)$ is used (B is a right A-module via f).

Proposition 15. The construction F in Definition 12 extends to a functor

$$F\colon \mathsf{Pm\text{-}Nom} \to \mathsf{Sb\text{-}Nom},$$

defined on an equivariant map $f\colon X \to Y$ by $F(f)([m, x]) = [m, f(x)] \in F(Y)$.

Proof. We first prove well-definedness and then the functoriality.

$F(X)$ is an Sb-set. To this end we check that the Sb-action is well-defined. Let $[m_1, x_1] = [m_2, x_2] \in F(X)$ and let $m \in \mathsf{Sb}$. By Lemma 13, there is some permutation g such that $gx_1 = x_2$ and $m_1|_C = m_2 g|_C$ for some support C of x_1. By post-composition with m we get $mm_1|_C = mm_2 g|_C$, which means (again by the lemma) that $[mm_1, x_1] = [mm_2, x_2]$. Thus $m[m_1, x_1] = m[m_2, x_2]$, which concludes well-definedness.

For associativity and unitality of the Sb-action, we simply note that it is directly defined by left multiplication of Sb which is associative and unital. This concludes that $F(X)$ is an Sb-set.

$F(X)$ is a nominal Sb set. Given an element $[m, x] \in F(X)$ and a Pm-support C of x, we will prove that $m \cdot C$ is an Sb-support for $[m, x]$. Suppose that we have $m_1, m_2 \in \mathsf{Sb}$ such that $m_1|_{m\cdot C} = m_2|_{m\cdot C}$. By pre-composition with m we get $m_1 m|_C = m_2 m|_C$ and this leads us to conclude $[m_1 m, x] = [m_2 m, x]$. So $m_1[m, x] = m_2[m, x]$ as required.

Functoriality. Let $f\colon X \to Y$ be a Pm-equivariant map. To see that $F(f)$ is well-defined consider $[m_1, x_1] = [m_2, x_2]$. By Lemma 13, there is a permutation g such that $gx_1 = x_2$ and $m_1|_C = m_2 g|_C$ for some support C of x_1. Applying $F(f)$ gives on one hand $[m_1, f(x_1)]$ and on the other hand $[m_2, f(x_2)] = [m_2, f(gx_1)] = [m_2, gf(x_1)] = [m_2 g, f(x_1)]$ (we used equivariance in the second step). Since $m_1|_C = m_2 g|_C$ and f preserves supports we have $[m_2 g, f(x_1)] = [m_1, f(x_1)]$.

For Sb-equivariance we consider both $n \cdot F(f)([m, x]) = n[m, f(x)] = [nm, f(x)]$ and $F(f)(n \cdot [m, x]) = F(f)([nm, x]) = [nm, f(x)]$. This shows that $nF(f)([m, x]) = F(f)(n[m, x])$ and concludes that we have a map $F(f)\colon F(X) \to F(Y)$.

The fact that F preserves the identity function and composition follows from the definition directly. □

Theorem 16. The functor F is left adjoint to U:

$$\mathsf{Pm\text{-}Nom} \underset{U}{\overset{F}{\rightleftarrows}} \mathsf{Sb\text{-}Nom} \qquad (\perp)$$

Proof. We show that, for every nominal set X, there is a map $\eta_X : X \to UF(X)$ with the necessary universal property: for every Pm-equivariant $f : X \to U(Y)$ there is a unique Sb-equivariant map $f^\sharp : FX \to Y$ such that $U(f^\sharp) \circ \eta_X = f$. Define η_X by $\eta_X(x) = [\mathrm{id}, x]$. This is equivariant: $g \cdot \eta_X(x) = g[\mathrm{id}, x] = [g, x] = [\mathrm{id}, gx] = \eta_X(gx)$. Now, for $f : X \to U(Y)$, define $f^\sharp([m, x]) = m \cdot f(x)$ for $x \in X$ and $m \in \mathsf{Sb}$. Then $U(f^\sharp) \circ \eta_X(x) = f^\sharp([\mathrm{id}, x]) = \mathrm{id} \cdot f(x) = f(x)$.

To show that $f^\sharp$ is well-defined, consider $[m_1, x_1] = [m_2, x_2]$ (we have to prove that $m_1 \cdot f(x_1) = m_2 \cdot f(x_2)$). By Lemma 13, there is a $g \in \mathsf{Pm}$ such that $gx_1 = x_2$ and $m_2 g|_C = m_1|_C$ for a Pm-support C of x_1. Now C is also a Pm-support for $f(x)$ and hence it is an Sb-support of $f(x)$ (Lemma 3). We conclude that $m_2 \cdot f(x_2) = m_2 \cdot f(gx_1) = m_2 g \cdot f(x_1) = m_1 \cdot f(x_1)$ (we use Pm-equivariance in the one but last step and Sb-support in the last step). Finally, Sb-equivariance of $f^\sharp$ and uniqueness are straightforward calculations. □

The counit $\epsilon : FU(Y) \to Y$ is defined by $\epsilon([m, x]) = m \cdot x$. For the inverse of $-^\sharp$, let $g : F(X) \to Y$ be an Sb-equivariant map; then $g^\flat : X \to U(Y)$ is given by $g^\flat(x) = g([\mathrm{id}, x])$. Note that the unit η is a Pm-equivariant map, hence it preserves supports (i.e., any support of x also supports $[\mathrm{id}, x]$). This also means that if C is a support of x, then $m \cdot C$ is a support of $[m, x]$ (by Lemma 5).

2.1 *On (separated) products*

The functor F not only preserves coproducts, being a left adjoint, but it also maps the separated product to products:

Theorem 17. The functor F is strong monoidal, from the monoidal category $(\mathsf{Pm}\text{-}\mathbf{Set}, *, 1)$ to $(\mathsf{Sb}\text{-}\mathbf{Set}, \times, 1)$. In particular, the map p given by

$$p = \langle F(\pi_1), F(\pi_2) \rangle : F(X * Y) \to F(X) \times F(Y)$$

is an isomorphism, natural in X and Y.

Proof. We prove that p is an isomorphism. It suffices to show that p is injective and surjective. Note that $p([m, (x, y)]) = ([m, x], [m, y])$.

Surjectivity. Let $([m_1, x], [m_2, y])$ be an element of $F(X) \times F(Y)$. We take an element $y' \in Y$ such that $y' \# \mathrm{supp}(x)$ and $y' = gy$ for some $g \in \mathsf{Pm}$. Now we have an element $(x, y') \in X * Y$. By Lemma 5, we have $\mathrm{supp}(y') = \mathrm{supp}(y)$. Define the map

$$m(x) = \begin{cases} m_1(x) & \text{if } x \in \mathrm{supp}(x) \\ m_2(g^{-1}(x)) & \text{if } x \in \mathrm{supp}(y') \\ x & \text{otherwise.} \end{cases}$$

(Observe that $\mathrm{supp}(x) \# \mathrm{supp}(y')$, so the cases are not overlapping.) The map m is an element of Sb. Now consider the element $z = [m, (x, y')] \in F(X * Y)$. Applying p to z

gives the element $([m, x], [m, y'])$. First, we note that $[m, x] = [m_1, x]$ by the definition of m. Second, we show that $[m, y'] = [m_2, y]$. Observe that $mg|_{supp(y)} = m_2|_{supp(y)}$ by definition of m. Since $supp(y)$ is a support of y, we have $[mg, y] = [m_2, y]$, and since $[mg, y] = [m, gy] = [m, y']$ we are done. Hence $p([m, (x, y')]) = ([m, x], [m, y']) = ([m_1, x], [m_2, y])$, so p is surjective.

Injectivity. Let $[m_1, (x_1, y_1)]$ and $[m_2, (x_2, y_2)]$ be two elements. Suppose that they are mapped to the same element, i.e., $[m_1, x_1] = [m_2, x_2]$ and $[m_1, y_1] = [m_2, y_2]$. Then there are permutations g_x, g_y such that $x_2 = g_x x_1$ and $y_2 = g_y y_1$. Moreover, let $C = supp(x_1)$ and $D = supp(y_1)$; then we have $m_1|_C = m_2 g_x|_C$ and $m_1|_D = m_2 g_y|_D$. In order to show the two original elements are equal, we have to provide a single permutation g. Define for, $z \in C \cup D$,

$$g_0(z) = \begin{cases} g_x(z) & \text{if } z \in C \\ g_y(z) & \text{if } z \in D. \end{cases}$$

(Again, C and D are disjoint.) The function g_0 is injective since the least supports of x_2 and y_2 are disjoint. Hence g_0 defines a local isomorphism from $C \cup D$ to $g_0(C \cup D)$. By homogeneity (Pitts, 2013), the map g_0 extends to a permutation $g \in \mathsf{Pm}$ with $g(z) = g_x(z)$ for $z \in C$ and $g(z) = g_y(z)$ for $z \in D$. In particular we get $(x_2, y_2) = g(x_1, y_1)$. We also obtain $m_1|_{C \cup D} = m_2 g|_{C \cup D}$. This proves that $[m_1, (x_1, y_1)] = [m_2, (x_2, y_2)]$, and so the map p is injective.

Unit and coherence. To show that F preserves the unit, we note that $[m, 1] = [m', 1]$ for every $m, m' \in \mathsf{Sb}$, as the empty set supports 1 and so $m|_\emptyset = m'|_\emptyset$ vacuously holds. We conclude $F(1)$ is a singleton. By the definition $p([m, (x, y)]) = ([m, x], [m, y])$, one can check the coherence axioms elementary. □

Since F also preserves coproducts (being a left adjoint), we obtain that F maps the set of separated words to the set of all words.

Corollary 18. For any Pm-nominal set X, we have $F(X^{(*)}) \cong (FX)^*$.

As we will show below, the functor F preserves the set $\mathbb{A}$ of atoms. This is an instance of a more general result about preservation of one-dimensional objects.

Lemma 19. The functors F and U are equivalences on $\leqslant 1$-dimensional objects. Concretely, for $X \in \mathsf{Pm}\text{-}\mathbf{Nom}$ and $Y \in \mathsf{Sb}\text{-}\mathbf{Nom}$:
- If $\dim(X) \leqslant 1$, then the unit $\eta : X \to UF(X)$ is an isomorphism.
- If $\dim(Y) \leqslant 1$, then the co-unit $\epsilon : FU(X) \to X$ is an isomorphism.

Before we prove this lemma, we need the following technical property of $\leqslant 1$-dimensional Sb-sets.

Lemma 20. Let Y be a nominal Sb-set. If an element $y \in Y$ is supported by a singleton set (or even the empty set), then

$$\{my \mid m \in \mathsf{Sb}\} = \{gy \mid g \in \mathsf{Pm}\}.$$

Proof. Let $y \in Y$ be supported by $\{a\}$ and let $m \in \mathsf{Sb}$. Now consider $b = m(a)$ and the bijection $g = (a\,b)$. Now $m|_{\{a\}} = g|_{\{a\}}$, meaning that $my = gy$. So the set $\{my \mid m \in \mathsf{Sb}\}$ is contained in $\{gy \mid g \in \mathsf{Pm}\}$. The inclusion the other way is trivial, which means $\{my \mid m \in \mathsf{Sb}\} = \{gy \mid g \in \mathsf{Pm}\}$. □

Proof of Lemma 19. It is easy to see that $\eta : x \mapsto [\mathrm{id}, x]$ is injective. Now to see that η is surjective, let $[m, x] \in \mathsf{UF}(X)$ and consider a support $\{a\}$ of x (this is a singleton or empty since $\dim(X) \leqslant 1$). Let $b = m(a)$ and consider the swap $g = (a\,b)$. Now $[m, x] = [mg^{-1}, gx]$ and note that $\{b\}$ supports gx and $mg^{-1}|_{\{b\}} = \mathrm{id}|_{\{b\}}$. We continue with $[mg^{-1}, gx] = [\mathrm{id}, gx]$, which concludes that gx is the preimage of $[m, x]$. Hence η is an isomorphism.

To see that $\epsilon : [m, y] \mapsto my$ is surjective, just consider $m = \mathrm{id}$. To see that ϵ is injective, let $[m, y], [m', y'] \in \mathsf{FU}(Y)$ be two elements such that $my = m'y'$. Then by using Lemma 20 we find $g, g' \in \mathsf{Pm}$ such that $gy = my = m'y' = g'y'$. This means that y and y' are in the same orbit (of $\mathsf{U}(Y)$) and have the same dimension. Case 1: $\mathrm{supp}(y) = \mathrm{supp}(y') = \emptyset$, then $[m, y] = [\mathrm{id}, y] = [\mathrm{id}, y'] = [m', y']$. Case 2: $\mathrm{supp}(y) = \{a\}$ and $\mathrm{supp}(y') = \{b\}$, then $\mathrm{supp}(gy) = \{g(a)\}$ (Lemma 5). In particular we now now that m and g map a to $c = g(a)$, likewise m' and g' map b to c. Now $[m, y] = [m, g^{-1}g'y'] = [mg^{-1}g', y'] = [m', y']$, where we used $mg^{-1}g(b) = c = m'(b)$ in the last step. This means that ϵ is injective and hence an isomorphism. □

By Lemma 19, we may consider the set $\mathbb{A}$ as both Sb-set and Pm-set (abusing notation). And we get an isomorphism $\mathsf{F}(\mathbb{A}) \cong \mathbb{A}$ of nominal Sb-sets. To appreciate the above results, we give a concrete characterisation of one-dimensional nominal sets:

Lemma 21. Let X be a nominal M-set, for $M \in \{\mathsf{Sb}, \mathsf{Pm}\}$. Then $\dim(X) \leqslant 1$ iff there exist (discrete) sets Y and I such that $X \cong Y + \coprod_I \mathbb{A}$.

In particular, the one-dimensional objects include the alphabets used for *data words*, consisting of a product $S \times \mathbb{A}$ of a discrete set S of action labels and the set of atoms. These alphabets are very common in the study of register automata (see, e.g., Isberner, et al., 2014).

By the above and Theorem 17, F maps separated powers of $\mathbb{A}$ to powers, and the set of separated words over $\mathbb{A}$ to the Sb-set of words over $\mathbb{A}$.

Corollary 22. We have $\mathsf{F}(\mathbb{A}^{(n)}) \cong \mathbb{A}^n$ and $\mathsf{F}(\mathbb{A}^{(*)}) \cong \mathbb{A}^*$.

2.2 *On exponents*

We have described how F and U interact with (separated) products. In this section, we establish a relationship between the magic wand $(-\!\!*)$ and the exponent of nominal Sb-sets $(\to^{\mathsf{Sb}}_{\mathsf{fs}})$.

Definition 23. Let $X \in \mathsf{Pm}\text{-}\mathbf{Nom}$ and $Y \in \mathsf{Sb}\text{-}\mathbf{Nom}$. We define a Pm-equivariant map ϕ as follows:

$$\phi : (X -\!\!* \mathsf{U}(Y)) \to \mathsf{U}(\mathsf{F}(X) \to^{\mathsf{Sb}}_{\mathsf{fs}} Y)$$

is defined by using the composition

$$\mathsf{F}(X -\!\!* \mathsf{U}(Y)) \times \mathsf{F}(X) \xrightarrow{p^{-1}} \mathsf{F}((X -\!\!* \mathsf{U}(Y)) * X) \xrightarrow{\mathsf{F}(ev)} \mathsf{FU}(Y) \xrightarrow{\epsilon} Y,$$

where p^{-1} is from Theorem 17 and ev is the evaluation map of the exponent $-\!\!*$. By Currying and the adjunction we arrive at ϕ:

$$\dfrac{\dfrac{\mathsf{F}(X -\!\!* \mathsf{U}(Y)) \times \mathsf{F}(X) \to Y}{\mathsf{F}(X -\!\!* \mathsf{U}(Y)) \to (\mathsf{F}(X) \to^{\mathsf{Sb}}_{\mathsf{fs}} Y)}\ \text{by Currying}}{\phi : (X -\!\!* \mathsf{U}(Y)) \to \mathsf{U}(\mathsf{F}(X) \to^{\mathsf{Sb}}_{\mathsf{fs}} Y)}\ \text{by Theorem 16}$$

With this map we can prove a generalisation of Theorem 16. In particular, the following theorem generalises the one-to-one correspondence between maps $X \to \mathsf{U}(Y)$ and maps $\mathsf{F}(X) \to Y$. First, it shows that this correspondence is Pm-equivariant. Second, it extends the correspondence to all finitely supported maps and not just the equivariant ones.

Theorem 24. The sets $X -\!\!* \mathsf{U}(Y)$ and $\mathsf{U}(\mathsf{F}(X) \to^{\mathsf{Sb}}_{\mathsf{fs}} Y)$ are naturally isomorphic via ϕ as nominal Pm-sets.

Proof. We define some additional maps in order to construct the inverse of ϕ. First, from Theorem 16 we get the following isomorphism:

$$q : \mathsf{U}(X \times Y) \xrightarrow{=} \mathsf{U}(X) \times \mathsf{U}(Y)$$

Second, with this map and Currying, we obtain the following two natural maps:

$$\dfrac{\mathsf{U}(\mathsf{F}(X) \to^{\mathsf{Sb}}_{\mathsf{fs}} Y) \times \mathsf{UF}(X) \xrightarrow{q^{-1}} \mathsf{U}((\mathsf{F}(X) \to^{\mathsf{Sb}}_{\mathsf{fs}} Y) \times \mathsf{F}(X)) \xrightarrow{\mathsf{U}(ev)} \mathsf{U}(Y)}{\alpha : \mathsf{U}(\mathsf{F}(X) \to^{\mathsf{Sb}}_{\mathsf{fs}} Y) \to (\mathsf{UF}(X) \to^{\mathsf{Pm}}_{\mathsf{fs}} \mathsf{U}(Y))}\ \text{by Currying}$$

$$\dfrac{(\mathsf{UF}(X) \to^{\mathsf{Pm}}_{\mathsf{fs}} \mathsf{U}(Y)) \times X \xrightarrow{\mathrm{id} \times \eta} (\mathsf{UF}(X) \to^{\mathsf{Pm}}_{\mathsf{fs}} \mathsf{U}(Y)) \times \mathsf{UF}(X) \xrightarrow{ev} \mathsf{U}(Y)}{\beta : (\mathsf{UF}(X) \to^{\mathsf{Pm}}_{\mathsf{fs}} \mathsf{U}(Y)) \to (X \to^{\mathsf{Pm}}_{\mathsf{fs}} \mathsf{U}(Y))}\ \text{by Currying}$$

Last, we note that the inclusion $A * B \subseteq A \times B$ induces a *restriction* map $r\colon (B \to^{\mathsf{Pm}}_{\mathsf{fs}} C) \to (B \multimap C)$ (again by Currying). A calculation shows that $r \circ \beta \circ \alpha$ is the inverse of ϕ. □

Note that this theorem gives an alternative characterisation of the magic wand in terms of the exponent in Sb-$\mathbf{Nom}$, if the codomain is $U(Y)$. Moreover, for a 1-dimensional object X in Sb-$\mathbf{Nom}$, we obtain the following special case of the theorem (using the co-unit isomorphism from Lemma 19):

Corollary 25. Let X, Y be nominal Sb-sets. For 1-dimensional X, the nominal Pm-set $U(X) \multimap U(Y)$ is naturally isomorphic to $U(X \to^{\mathsf{Sb}}_{\mathsf{fs}} Y)$.

Remark 26. The set $\mathbb{A} \multimap U(X)$ coincides with the atom abstraction $[\mathbb{A}]UX$ (Remark 10). Hence, as a special case of Corollary 25, we recover Theorem 34 of Gabbay and Hofmann (2008), which states a bijective correspondence between $[\mathbb{A}]UX$ and $U(\mathbb{A} \to^{\mathsf{Sb}}_{\mathsf{fs}} X)$.

3 *Nominal and separated automata*

In this section, we study nominal automata, which recognise languages over infinite alphabets. After recalling the basic definitions, we introduce a new variant of automata based on the separating product, which we call *separated nominal automata*. These automata represent nominal languages which are Sb-equivariant, essentially meaning they are closed under substitution. Our main result is that, if a 'classical' nominal automaton (over Pm) represents a language L which is Sb-equivariant, then L can also be represented by a separated nominal automaton. The latter can be exponentially smaller (in number of orbits) than the original automaton, as we show in a concrete example.

Remark 27. We will work with a general output set O instead of just acceptance. The reason for this is that Sb-equivariant functions $L\colon \mathbb{A} \to 2$ are not very interesting: they are defined purely by the length of the input. By using more general output O, we may still capture interesting behaviour, e.g., the automaton in Example 29.

Definition 28. Let Σ, O be Pm-sets, called input/output alphabet respectively.

- A (Pm)*-nominal language* is an equivariant map of the form $L\colon \Sigma^* \to O$.
- A *nominal* (*Moore*) *automaton* $\mathcal{A} = (Q, \delta, o, q_0)$ consists of a nominal set of states Q, an equivariant transition function $\delta\colon Q \times \Sigma \to Q$, an equivariant output function $o\colon Q \to O$, and an initial state $q_0 \in Q$ with an empty support.
- The *language semantics* is the map $l\colon Q \times \Sigma^* \to O$, defined inductively by

$$l(x, \varepsilon) = o(x)\,, \qquad l(x, aw) = l(\delta(x, a), w)$$

for all $x \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$.
- For $l^\flat : Q \to (\Sigma^* \to^{\mathsf{Pm}}_{\mathsf{fs}} O)$ the transpose of l, we have that $l^\flat(q_0) : \Sigma^* \to O$ is equivariant; this is called the *language accepted by* $\mathcal{A}$.

Note that the language accepted by an automaton can equivalently be characterised by considering paths through the automaton from the initial state.

If the state space Q and the alphabets Σ, O are orbit finite, this allows us to run algorithms (reachability, minimization, etc.) on such automata, but there is no need to assume this for now. For an automaton $\mathcal{A} = (Q, \delta, o, q_0)$, we define the set of *reachable states* as the least set $R(\mathcal{A}) \subseteq Q$ such that $q_0 \in R(\mathcal{A})$ and for all $x \in R(\mathcal{A})$ and $a \in \Sigma$, $\delta(x, a) \in R(\mathcal{A})$.

Example 29. We model a bounded FIFO queue of size n as a nominal Moore automaton, explicitly handling the data in the automaton structure.[27] The input alphabet Σ and output alphabet O are as follows:

$$\Sigma = \{\mathtt{Put}(a) \mid a \in \mathbb{A}\} \cup \{\mathtt{Pop}\}, \qquad O = \mathbb{A} \cup \{\bot\}.$$

The input alphabet encodes two actions: putting a new value on the queue and popping a value. The output is either a value (the front of the queue) or $\bot$ if the queue is empty. A queue of size n is modelled by the automaton (Q, δ, o, q_0) defined as follows.

$$Q = \mathbb{A}^{\leqslant n} \cup \{\bot\}, \qquad q_0 = \epsilon, \qquad o(a_1 \ldots a_k) = \begin{cases} a_1 & \text{if } k \geqslant 1 \\ \bot & \text{otherwise} \end{cases}$$

$$\delta(a_1 \ldots a_k, \mathtt{Put}(b)) = \begin{cases} a_1 \ldots a_k b & \text{if } k < n \\ \bot & \text{otherwise} \end{cases} \qquad \delta(a_1 \ldots a_k, \mathtt{Pop}) = \begin{cases} a_2 \ldots a_k & \text{if } k > 0 \\ \bot & \text{otherwise} \end{cases}$$

$$\delta(\bot, x) = \bot$$

The automaton is depicted in Figure 7.1 for the case $n = 3$. The language accepted by this automaton assigns to a word w the first element of the queue after executing the instructions in w from left to right, and $\bot$ if the input is ill-behaved, i.e., $\mathtt{Pop}$ is applied to an empty queue or $\mathtt{Put}(a)$ to a full queue.

Definition 30. Let Σ, O be Pm-sets. A *separated language* is an equivariant map of the form $\Sigma^{(*)} \to O$. A *separated automaton* $\mathcal{A} = (Q, \delta, o, q_0)$ consists of Q, o and q_0 defined as in a nominal automaton, and an equivariant transition function $\delta : Q * \Sigma \to Q$.

The *separated language semantics* of such an automaton is given by the map $s : Q * \Sigma^{(*)} \to O$, defined by

$$s(x, \epsilon) = o(x), \qquad s(x, aw) = s(\delta(x, a), w)$$

[27] We use a reactive version of the queue data structure which is slightly different from the versions of Isberner, et al. (2014) and Moerman, et al. (2017).

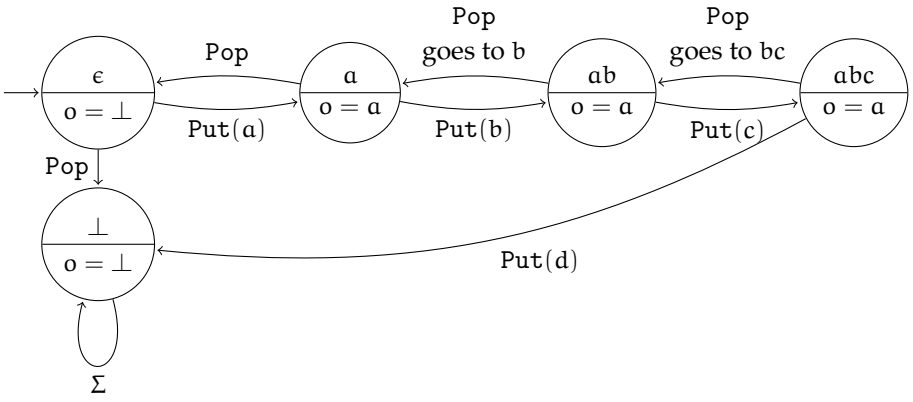


Figure 7.1 The FIFO automaton from Example 29 with $n = 3$. The right-most state consists of *five* orbits as we can take a, b, c distinct, all the same, or two of them equal in three different ways. Consequently, the complete state space has ten orbits. The output of each state is denoted in the lower part.

for all $x \in Q$, $a \in \Sigma$ and $w \in \Sigma^{(*)}$ such that $x \# aw$ and $a \# w$.

Let $s^\flat : Q \to (\Sigma^{(*)} \multimap O)$ be the transpose of s. Then $s^\flat(q_0) : \Sigma^{(*)} \to O$ corresponds to a separated language, this is called the *separated language accepted by* $\mathcal{A}$.

By definition of the separated product, the transition function is only defined on a state x and letter $a \in \Sigma$ if $x \# a$. In Example 36 below, we describe the bounded FIFO as a separated automaton, and describe its accepted language.

First, we show how the language semantics of separated nominal automata extends to a language over *all* words, provided that both the input alphabet Σ and the output alphabet O are Sb-sets.

Definition 31. Let Σ and O be nominal Sb-sets. An Sb-equivariant function $L : \Sigma^* \to O$ is called an *Sb-language*.

Notice the difference between an Sb-language $L : \Sigma^* \to O$ and a Pm-language $L' : (U\Sigma)^* \to U(O)$. They are both functions from Σ^* to O, but the latter is only Pm-equivariant, while the former satisfies the stronger property of Sb-equivariance. Languages over separated words, and Sb-languages, are connected as follows.

Theorem 32. Suppose Σ, O are both nominal Sb-sets, and suppose $\dim(\Sigma) \leqslant 1$. There is a one-to-one correspondence

$$\frac{S : (U\Sigma)^{(*)} \to UO \quad \text{Pm-equivariant}}{\overline{S} : \Sigma^* \to O \quad \text{Sb-equivariant}}$$

between separated languages and Sb-nominal languages. From $\overline{S}$ to S, this is given by application of the forgetful functor and restricting to the subset of separated words.

For the converse direction, given $w = a_1 ... a_n \in \Sigma^*$, let $b_1, ..., b_n \in \Sigma$ such that $w \# b_i$ for all i, and $b_i \# b_j$ for all i, j with $i \neq j$. Define $m \in \mathsf{Sb}$ by

$$m(a) = \begin{cases} a_i & \text{if } a = b_i \text{ for some } i \\ a & \text{otherwise} \end{cases}$$

Then $\overline{S}(a_1 a_2 a_3 \cdots a_n) = m \cdot S(b_1 b_2 b_3 \cdots b_n)$.

Proof. There is the following chain of one-to-one correspondences, from the results of the previous section:

$$\frac{\frac{\frac{(U\Sigma)^{(*)} \to UO}{F(U\Sigma)^{(*)} \to O}\ \text{by Theorem 16}}{(FU\Sigma)^* \to O}\ \text{by Corollary 18}}{\Sigma^* \to O}\ \text{by Lemma 19}$$

□

Thus, every separated automaton over $U(\Sigma), U(O)$ gives rise to an Sb-language $\overline{S}$, corresponding to the language S accepted by the automaton.

Any nominal automaton $\mathcal{A}$ restricts to a separated automaton, formally described in Definition 33. It turns out that if the (Pm)-language accepted by $\mathcal{A}$ is actually an Sb-language, then the restricted automaton already represents this language, as the extension $\overline{S}$ of the associated separated language S (Theorem 34). Hence, in such a case, the restricted separated automaton suffices to describe the language of $\mathcal{A}$.

Definition 33. Let $i : Q * U(\Sigma) \to Q \times U(\Sigma)$ be the natural inclusion map. A nominal automaton $\mathcal{A} = (Q, \delta, o, q_0)$ induces a separated automaton $\mathcal{A}_*$, by setting

$$\mathcal{A}_* = (Q, \delta \circ i, o, q_0).$$

Theorem 34. Suppose Σ, O are both Sb-sets, and suppose $\dim(\Sigma) \leqslant 1$. Let $L : (U\Sigma)^* \to UO$ be the Pm-nominal language accepted by a nominal automaton $\mathcal{A}$, and suppose L is Sb-equivariant. Let S be the separated language accepted by $\mathcal{A}_*$. Then $L = U(\overline{S})$.

Proof. If follows from the one-to-one correspondence in Theorem 32: on the bottom there are two languages (L and $U(\overline{S})$), while there is only the restriction of L on the top. We conclude that $L = U(\overline{S})$. □

As we will see in Example 36, separated automata allow us to represent Sb-languages in a much smaller way than nominal automata. Given a nominal automaton $\mathcal{A}$, a smaller separated automaton can be obtained by computing the reachable part of the restriction $\mathcal{A}_*$. The reachable part is defined similarly (but only where δ is defined) and denoted by $R(\mathcal{A}_*)$ as well.

Proposition 35. For any nominal automaton $\mathcal{A}$, we have $\mathsf{R}(\mathcal{A}_*) \subseteq \mathsf{R}(\mathcal{A})$.

The converse inclusion of the above proposition does certainly not hold, as shown by the following example.

Example 36. Let $\mathcal{A}$ be the automaton modelling a bounded FIFO queue (for some n), from Example 29. The Pm-nominal language L accepted by $\mathcal{A}$ is Sb-equivariant: it is closed under application of arbitrary substitutions.

The separated automaton $\mathcal{A}_*$ is given simply by restricting the transition function to $Q * \Sigma$, i.e., a $\mathtt{Put}(a)$-transition from a state $w \in Q$ exists only if a does not occur in w. The separated language S accepted by this new automaton is the restriction of the nominal language of $\mathcal{A}$ to separated words. By Theorem 34, we have $L = U(\overline{S})$. Hence, the separated automaton $\mathcal{A}_*$ represents L, essentially by closing the associated separated language S under all substitutions.

The *reachable* part of $\mathcal{A}_*$ is given by

$$\mathsf{R}_{\mathcal{A}_*} = \mathbb{A}^{(\leqslant n)} \cup \{\bot\}.$$

Clearly, restricting $\mathcal{A}_*$ to the reachable part does not affect the accepted language. However, while the orginal state space Q has exponentially many orbits in n, $\mathsf{R}_{\mathcal{A}_*}$ has only $n+1$ orbits! Thus, taking the reachable part of $\mathsf{R}_{\mathcal{A}_*}$ yields a separated automaton which represents the FIFO language L in a much smaller way than the original automaton.

3.1 *Separated automata: coalgebraic perspective*

Nominal automata and separated automata can be presented as *coalgebras* on the category of Pm-nominal sets. In this section we revisit the above results from this perspective, and generalise from (equivariant) languages to finitely supported languages. In particular, we retrieve the extension from separated languages to Sb-languages, by establishing Sb-languages as a final separated automaton. The latter result follows by instantiating a well-known technique for lifting adjunctions to categories of coalgebras, using the results of Section 2. In the remainder of this section we assume familiarity with the theory of coalgebras, see, e.g., Jacobs (2016) and Rutten (2000).

Definition 37. Let M be a submonoid of Sb, and let Σ, O be nominal M-sets, referred to as the input and output alphabet respectively. We define the functor $\mathsf{B}_M : M\text{-}\mathbf{Nom} \to M\text{-}\mathbf{Nom}$ by $\mathsf{B}_M(X) = O \times (\Sigma \to_{\mathrm{fs}}^{M} X)$. An (M)-*nominal* (*Moore*) *automaton* is a B_M-coalgebra.

A B_M-coalgebra can be presented as a nominal set Q together with the pairing

$$\langle o, \delta^\flat \rangle : Q \to O \times (\Sigma \to_{\mathrm{fs}}^{M} Q)$$

of an equivariant *output* function $o : Q \to O$, and (the transpose of) an equivariant *transition* function $\delta : Q \times \Sigma \to Q$. In case $M = \mathsf{Pm}$, this coincides with the automata

of Definition 28, omitting initial states. The language semantics is generalised accordingly, as follows. Given such a B_M-coalgebra $(Q, \langle o, \delta^\flat \rangle)$, the *language semantics* $l \colon Q \times \Sigma^* \to O$ is given by

$$l(x, \varepsilon) = o(x), \qquad l(x, aw) = l(\delta(x, a), w)$$

for all $x \in S$, $a \in \Sigma$ and $w \in \Sigma^*$.

Theorem 38. Let M be a submonoid of Sb, let Σ, O be nominal M-sets. The nominal M-set $\Sigma^* \to^M_{\mathrm{fs}} O$ extends to a final B_M-coalgebra $(\Sigma^* \to^M_{\mathrm{fs}} O, \zeta)$, such that the unique homomorphism from a given B_M-coalgebra is the transpose $l^\flat$ of the language semantics.

A *separated automaton* (Definition 30, without initial states) corresponds to a coalgebra for the functor $\mathsf{B}_* \colon \mathsf{Pm\text{-}Nom} \to \mathsf{Pm\text{-}Nom}$ given by $\mathsf{B}_*(X) = O \times (\Sigma \multimap X)$. The separated language semantics arises by finality.

Theorem 39. The set $\Sigma^{(*)} \multimap O$ is the carrier of a final B_*-coalgebra, such that the unique coalgebra homomorphism from a given B_*-coalgebra $(Q, \langle o, \delta \rangle)$ is the transpose $s^\flat$ of the separated language semantics $s \colon Q * \Sigma^{(*)} \to O$ (Definition 30).

Next, we provide an alternative final B_*-coalgebra which assigns Sb-nominal languages to states of separated nominal automata. The essence is to obtain a final B_*-coalgebra from the final B_{Sb}-coalgebra. In order to prove this, we use a technique to lift adjunctions to categories of coalgebras. This technique occurs regularly in the coalgebraic study of automata (Jacobs, et al., 2015; Kerstan, et al., 2014; Klin & Rot, 2016).

Theorem 40. Let Σ be a Pm-set, and O an Sb-set. Define B_* and B_{Sb} accordingly, as $\mathsf{B}_*(X) = UO \times (\Sigma \multimap X)$ and $\mathsf{B}_{\mathsf{Sb}}(X) = O \times (F\Sigma \to^{\mathsf{Sb}}_{\mathrm{fs}} X)$.

There is an adjunction $\overline{F} \dashv \overline{U}$ in:

$$\mathsf{CoAlg}(\mathsf{B}_*) \underset{\overline{U}}{\overset{\overline{F}}{\rightleftarrows}} \mathsf{CoAlg}(\mathsf{B}_{\mathsf{Sb}}) \qquad (\perp)$$

where $\overline{F}$ and $\overline{U}$ coincide with F and U respectively on carriers.

Proof. There is a natural isomorphism $\lambda \colon \mathsf{B}_* U \to U \mathsf{B}_{\mathsf{Sb}}$ given by

$$\lambda \colon UO \times (\Sigma \multimap UX) \xrightarrow{\mathrm{id} \times \phi} UO \times U(F\Sigma \to^{\mathsf{Sb}}_{\mathrm{fs}} X) \xrightarrow{\cong} U(O \times (F\Sigma \to^{\mathsf{Sb}}_{\mathrm{fs}} X)),$$

where ϕ is the isomorphism from Theorem 24 and the isomorphism on the right comes from U being a right adjoint. The result now follows from Theorem 2.14 of Hermida and Jacobs (1998). In particular, $\overline{U}(X, \gamma) = (UX, \lambda^{-1} \circ U(\gamma))$. □

Since right adjoints preserve limits, and final objects in particular, we obtain the following. This gives an Sb-semantics of separated automata through finality.

Corollary 41. Let $((F\Sigma)^* \to^{Sb}_{fs} O, \zeta)$ be the final B_{Sb}-coalgebra (Theorem 38). The B_*-coalgebra $\overline{U}(\Sigma^* \to^{Sb}_{fs} O, \zeta)$ is final and carried by the set $(F\Sigma)^* \to^{Sb}_{fs} O$ of Sb-nominal languages.

4 *Related and future work*

Fiore and Turi (2001) described a similar adjunction between certain presheaf categories. However, Staton (2007) describes in his thesis that the usage of presheaves allows for many degenerate models and one should look at sheaves instead. The category of sheaves is equivalent to the category of nominal sets. Staton transfers the adjunction of Fiore and Turi to the sheaf categories. We conjecture that the adjunction presented in this paper is equivalent, but defined in more elementary means. The monoidal property of F, which is crucial for our application in automata, has not been discussed before.

An interesting line of research is the generalisation to other symmetries by Bojańczyk, et al. (2014). In particular, the total order symmetry is relevant, since it allows one to compare elements on their order, as often used in data words. In this case the symmetries are given by the group of all monotone bijections. Many results of nominal sets generalise to this symmetry. For monotone substitutions, however, the situation seems more subtle. For example, we note that a substitution which maps two values to the same value actually maps *all* the values in between to that value. Whether the adjunction from Theorem 16 generalises to other symmetries is left as future work.

This research was motivated by learning nominal automata. If we know a nominal automaton recognises an Sb-language, then we are better off learning a separated automaton directly. From the Sb-semantics of separated automata, it follows that we have a Myhill-Nerode theorem, which means that learning is feasible. We expect that this can be useful, since we can achieve an exponential reduction this way.

Bojańczyk, et al. (2014) prove that nominal automata are equivalent to register automata in terms of expressiveness. However, when translating from register automata with n states to nominal automata, we may get exponentially many orbits. This happens for instance in the FIFO automaton (Example 29). We have shown that the exponential blow-up is avoidable by using separated automata, for this example and in general for Sb-equivariant languages. An open problem is whether the latter requirement can be relaxed, by adding separated transitions only locally in a nominal automaton.

A possible step in this direction is to consider the monad $T = UF$ on Pm-Nom and incorporate it in the automaton model. We believe that this is the hypothesised "substitution monad" from Chapter 5. The monad is monoidal (sending separated products to Cartesian products) and if X is an orbit-finite nominal set, then so is $T(X)$.

This means that we can consider nominal T-automata and we can perhaps determinise them using coalgebraic methods (Silva, et al., 2013).

Acknowledgements

We would like to thank Gerco van Heerdt for his useful comments.

Bibliography

Aarts, F. D. (2014). *Tomte: bridging the gap between active learning and real-world systems*. (PhD thesis). Radboud University, Nijmegen, The Netherlands. Retrieved from http://hdl.handle.net/2066/130428 (5 and 42)

Aarts, F., de Ruiter, J., & Poll, E. (2013). Formal Models of Bank Cards for Free. In *ICST, Workshops Proceedings*. IEEE Computer Society. doi:10.1109/ICSTW.2013.60 (4)

Aarts, F., Fiterău-Broștean, P., Kuppens, H., & Vaandrager, F. W. (2015). Learning Register Automata with Fresh Value Generation. In *Theoretical Aspects of Computing - ICTAC - 12th International Colloquium Proceedings*. Springer. doi:10.1007/978-3-319-25150-9_11 (76, 103–104, 106, and 110)

Aarts, F., Jonsson, B., Uijen, J., & Vaandrager, F. W. (2015). Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, *46*(1), 1–41. doi:10.1007/s10703-014-0216-x (43 and 48)

Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F. W., & Verwer, S. (2014). Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, *96*(1-2), 189–224. doi:10.1007/s10994-013-5405-0 (42)

Aarts, F. & Vaandrager, F. W. (2010). Learning I/O Automata. In *CONCUR - Concurrency Theory, 21th International Conference Proceedings*. Springer. doi:10.1007/978-3-642-15375-4_6 (76)

Angluin, D. (1987). Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, *75*(2), 87–106. doi:10.1016/0890-5401(87)90052-6 (2, 11, 37, 42, 76, 78, 88, and 124–125)

Atiyah, M. F. & MacDonald, I. G. (1969). *Introduction to commutative algebra*. Addison-Wesley-Longman. (137)

Bastian, M., Heymann, S., & Jacomy, M. (2009). Gephi: An Open Source Software for Exploring and Manipulating Networks. In *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM*. The AAAI Press. Retrieved from http://aaai.org/ocs/index.php/ICWSM/09/paper/view/154 (55)

Behrmann, G., David, A., Larsen, K. G., Håkansson, J., Pettersson, P., Yi, W., & Hendriks, M. (2006). UPPAAL 4.0. In *Third International Conference on the Quantitative Evaluation of Systems (QEST)*. IEEE Computer Society. doi:10.1109/QEST.2006.59 (52)

Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., & Steffen, B. (2005). On the Correspondence Between Conformance Testing and Regular Inference. In *FASE, ETAPS, Proceedings*. Springer. doi:10.1007/978-3-540-31984-9_14 (37, 42, and 44)

Berg, T., Jonsson, B., & Raffelt, H. (2006). Regular Inference for State Machines with Parameters. In *FASE, ETAPS, Proceedings*. Springer. doi:10.1007/11693017_10 (106)

——— (2008). Regular Inference for State Machines Using Domains with Equality Tests. In *FASE, ETAPS, Proceedings*. Springer. doi:10.1007/978-3-540-78743-3_24 (106)

Bernhard, P. J. (1994). A Reduced Test Suite for Protocol Conformance Testing. *ACM Trans. Softw. Eng. Methodol.*, *3*(3), 201–220. doi:10.1145/196092.196088 (26)

Bojańczyk, M. (2018). *Slightly Infinite Sets*. Draft December 4, 2018. Retrieved from https://www.mimuw.edu.pl/~bojan/upload/main-6.pdf (9 and 132)

Bojańczyk, M., Braud, L., Klin, B., & Lasota, S. (2012). Towards nominal computation. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM. doi:10.1145/2103656.2103704 (101 and 127)

Bojańczyk, M., Klin, B., & Lasota, S. (2014). Automata theory in nominal sets. *Logical Methods in Computer Science*, *10*(3). doi:10.2168/LMCS-10(3:4)2014 (9, 76, 81–82, 84, 86–87, 104, 108, 110–113, 120, 127, 132, 135, and 149)

Bojańczyk, M., Klin, B., Lasota, S., & Toruńczyk, S. (2013). Turing Machines with Atoms. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*. Author. doi:10.1109/LICS.2013.24 (132)

Bojańczyk, M. & Lasota, S. (2012). A Machine-Independent Characterization of Timed Languages. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP, Proceedings, Part II*. Springer. doi:10.1007/978-3-642-31585-5_12 (76, 111, and 128)

Bollig, B., Habermehl, P., Kern, C., & Leucker, M. (2008). Angluin-Style Learning of NFA. Retrieved from http://www.lsv.fr/Publis/RAPPORTS_LSV/PDF/rr-lsv-2008-28.pdf (Research Report LSV-08-28, LSV, ENS Cachan) (99)

——— (2009). Angluin-Style Learning of NFA. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI*. Author. Retrieved from http://ijcai.org/Proceedings/09/Papers/170.pdf (83, 93–95, 98–99, and 105)

Bollig, B., Habermehl, P., Leucker, M., & Monmege, B. (2013). A Fresh Approach to Learning Register Automata. In *Developments in Language Theory - 17th International Conference, DLT, Proceedings*. Springer. doi:10.1007/978-3-642-38771-5_12 (5, 76, 103, 106, and 110)

Bonchi, F. & Pous, D. (2013). Checking NFA equivalence with bisimulations up to congruence. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM. doi:10.1145/2429069.2429124 (60)

——— (2015). Hacking nondeterminism with induction and coinduction. *Commun. ACM*, *58*(2), 87–95. doi:10.1145/2713167 (36 and 103)

Botincan, M. & Babic, D. (2013). Sigma*: symbolic learning of input-output specifications. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM. doi:10.1145/2429069.2429123 (106)

Cameron, P. J., Solomon, R., & Turull, A. (1989). Chains of subgroups in symmetric groups. *Journal of algebra*, *127*(2), 340–352. doi:10.1016/0021-8693(89)90256-1 (92)

Cassel, S. (2015). *Learning Component Behavior from Tests: Theory and Algorithms for Automata with Data*. (PhD thesis). Uppsala University, Sweden. Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-265369 (5)

Cassel, S., Howar, F., Jonsson, B., Merten, M., & Steffen, B. (2015). A succinct canonical register automaton model. *J. Log. Algebr. Meth. Program.*, *84*(1), 54–66. doi:10.1016/j.jlamp.2014.07.004 (42)

Cassel, S., Howar, F., Jonsson, B., & Steffen, B. (2016). Active learning for extended finite state machines. *Formal Asp. Comput.*, *28*(2), 233–263. doi:10.1007/s00165-016-0355-5 (76, 106, and 110)

Chalupar, G., Peherstorfer, S., Poll, E., & de Ruiter, J. (2014). Automated Reverse Engineering using Lego®. In *8th USENIX Workshop on Offensive Technologies, WOOT*. USENIX Association. Retrieved from https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar (4)

Chan, W. Y. L., Vuong, S. T., & Ito, M. R. (1989). An Improved Protocol Test Generation Procedure Based on UIOs. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols, SIGCOMM*. ACM. doi:10.1145/75246.75274 (26 and 28)

Cho, C. Y., Babic, D., Shin, E. C. R., & Song, D. (2010). Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS*. ACM. doi:10.1145/1866307.1866355 (42)

Chow, T. S. (1978). Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Software Eng.*, *4*(3), 178–187. doi:10.1109/TSE.1978.231496 (6, 26, 37, 43, and 49)

Ciancia, V., Kurz, A., & Montanari, U. (2010). Families of Symmetries as Efficient Models of Resource Binding. *Electr. Notes Theor. Comput. Sci.*, *264*(2), 63–81. doi:10.1016/j.entcs.2010.07.014 (127)

Ciancia, V. & Montanari, U. (2010). Symmetries, local names and dynamic (de)-allocation of names. *Inf. Comput.*, *208*(12), 1349–1367. doi:10.1016/j.ic.2009.10.007 (107 and 127)

Clouston, R. (2013). Generalised Name Abstraction for Nominal Sets. In *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS, Proceedings*. Author. doi:10.1007/978-3-642-37075-5_28 (136)

D'Antoni, L. & Veanes, M. (2014). Minimization of symbolic automata. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM. doi:10.1145/2535838.2535849 (76)

——— (2017). The Power of Symbolic Automata and Transducers. In *Computer Aided Verification - 29th International Conference, CAV Proceedings, Part I*. Springer. doi:10.1007/978-3-319-63387-9_3 (110 and 128)

David, A., Möller, M. O., & Yi, W. (2002). Formal Verification of UML Statecharts with Real-Time Extensions. In *FASE, ETAPS, Proceedings*. Springer. doi:10.1007/3-540-45923-5_15 (52)

Demri, S. & Lazic, R. (2009). LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, *10*(3), 16:1–16:30. doi:10.1145/1507244.1507246 (81)

Denis, F., Lemay, A., & Terlutte, A. (2002). Residual Finite State Automata. *Fundam. Inform.*, *51*(4), 339–368. Retrieved from http://content.iospress.com/articles/fundamenta-informaticae/fi51-4-02 (93)

Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A. R., & Yevtushenko, N. (2010). FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information & Software Technology*, *52*(12), 1286–1297. doi:10.1016/j.infsof.2010.07.001 (21, 31, 35, 39, 60, and 70–72)

Dorofeeva, R., El-Fakih, K., & Yevtushenko, N. (2005). An Improved Conformance Testing Method. In *Formal Techniques for Networked and Distributed Systems - FORTE, 25th IFIP WG 6.1 International Conference, Proceedings*. Springer. doi:10.1007/11562436_16 (38)

Drews, S. & D'Antoni, L. (2017). Learning Symbolic Automata. In *TACAS, ETAPS, Proceedings, Part I*. Author. doi:10.1007/978-3-662-54577-5_10 (110)

de la Higuera, C. (2010). *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press. doi:10.1017/CBO9781139194655 (42)

de Moura, L. M. & Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *TACAS, ETAPS, Proceedings*. Springer. doi:10.1007/978-3-540-78800-3_24 (103)

de Ruiter, J. & Poll, E. (2015). Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium, USENIX Security*. USENIX Association. Retrieved from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter (4)

Emerson, E. A. & Sistla, A. P. (1996). Symmetry and Model Checking. *Formal Methods in System Design*, *9*(1/2), 105–131. doi:10.1007/BF00625970 (9)

Endo, A. T. & Simão, A. (2013). Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods. *Information & Software Technology*, *55*(6), 1045–1062. doi:10.1016/j.infsof.2013.01.001 (39)

Eshuis, R., Jansen, D. N., & Wieringa, R. (2002). Requirements-Level Semantics and Model Checking of Object-Oriented Statecharts. *Requir. Eng.*, *7*(4), 243–263. doi:10.1007/s007660200019 (46)

Ferrari, G. L., Montanari, U., & Tuosto, E. (2005). Coalgebraic minimization of HD-automata for the π-calculus using polymorphic types. *Theor. Comput. Sci.*, *331*(2-3), 325–365. doi:10.1016/j.tcs.2004.09.021 (9 and 127)

Fiore, M. P. & Turi, D. (2001). Semantics of Name and Value Passing. In *16th Annual IEEE Symposium on Logic in Computer Science, Proceedings*. IEEE Computer Society. doi:10.1109/LICS.2001.932486 (149)

Fiterău-Broștean, P. (2018). *Active Model Learning for the Analysis of Network Protocols*. (PhD thesis). Radboud University, Nijmegen, The Netherlands. Retrieved from http://hdl.handle.net/2066/187331 (5 and 14)

Fiterău-Broștean, P. & Howar, F. (2017). Learning-Based Testing the Sliding Window Behavior of TCP Implementations. In *Critical Systems: Formal Methods and Automated Verification, Joint FMICS-AVoCS, Proceedings*. Springer. doi:10.1007/978-3-319-67113-0_12 (4)

Fiterău-Broștean, P., Janssen, R., & Vaandrager, F. W. (2014). Learning Fragments of the TCP Network Protocol. In *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS, Proceedings*. Springer. doi:10.1007/978-3-319-10702-8_6 (42)

——— (2016). Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification - 28th International Conference, CAV, Proceedings, Part II*. Springer. doi:10.1007/978-3-319-41540-6_25 (4 and 110)

Fiterău-Broștean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F. W., & Verleg, P. (2017). Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium*. ACM. doi:10.1145/3092282.3092289 (4)

Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., & Ghedamsi, A. (1991). Test Selection Based on Finite State Models. *IEEE Trans. Software Eng.*, *17*(6), 591–603. doi:10.1109/32.87284 (27, 43, and 50)

Gabbay, M. J. (2007). *Nominal Renaming Sets*. (Technical report). Author. Retrieved from https://www.gabbay.org/paper.html#nomrs-tr (Heriot-Watt University) (132 and 135)

Gabbay, M. J. & Hofmann, M. (2008). Nominal Renaming Sets. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR, Proceedings*. Author. doi:10.1007/978-3-540-89439-1_11 (132–136 and 143)

Gabbay, M. & Pitts, A. M. (1999). A New Approach to Abstract Syntax Involving Binders. In *14th Annual IEEE Symposium on Logic in Computer Science*. Author. doi:10.1109/LICS.1999.782617 (132)

——— (2002). A New Approach to Abstract Syntax with Variable Binding. *Formal Asp. Comput.*, *13*(3-5), 341–363. doi:10.1007/s001650200016 (9 and 110)

Garavel, H., Lang, F., Mateescu, R., & Serwe, W. (2011). CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *TACAS, ETAPS, Proceedings*. Springer. doi:10.1007/978-3-642-19835-9_33 (52 and 55)

Gill, A. (1962). *Introduction to the theory of finite-state machines*. McGraw-Hill. (60)

Graaf, B. & van Deursen, A. (2007). Model-Driven Consistency Checking of Behavioural Specifications. In *Model-based Methodologies for Pervasive and Embedded Software, 4th International Workshop, MOMPES, Proceedings*. IEEE Computer Society. doi:10.1109/MOMPES.2007.12 (43)

Gries, D. (1973). Describing an Algorithm by Hopcroft. *Acta Inf.*, *2*, 97–109. (61–62)

Grigore, R. & Tzevelekos, N. (2016). History-Register Automata. *Logical Methods in Computer Science*, *12*(1). doi:10.2168/LMCS-12(1:7)2016 (110 and 128)

Groz, R., Bremond, N., & Simão, A. (2018). Inferring FSM Models of Systems Without Reset. In *International Conference on Grammatical Inference, ICGI*. Proceedings of Machine Learning Research. (To appear) (39)

Groz, R., Li, K., Petrenko, A., & Shahbaz, M. (2008). Modular System Verification by Inference, Testing and Reachability Analysis. In *TestCom/FATES, Proceedings*. Springer. doi:10.1007/978-3-540-68524-1_16 (42)

Hansen, H. H., Ketema, J., Luttik, B., Mousavi, M. R., van de Pol, J., & dos Santos, O. M. (2010). Automated Verification of Executable UML Models. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO, Revised Papers*. Springer. doi:10.1007/978-3-642-25271-6_12 (52)

Hermida, C. & Jacobs, B. (1998). Structural Induction and Coinduction in a Fibrational Setting. *Inf. Comput.*, *145*(2), 107–152. doi:10.1006/inco.1998.2725 (148)

Hierons, R. M. & Türker, U. C. (2015). Incomplete Distinguishing Sequences for Finite State Machines. *Comput. J.*, *58*(11), 3089–3113. doi:10.1093/comjnl/bxv041 (39 and 71)

Hopcroft, J. E. (1971). An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations - Proceedings of an International Symposium on the Theory of Machines and Computations*. Academic Press. doi:10.1016/B978-0-12-417750-5.50022-1 (11, 60, 66, and 71)

Howar, F., Steffen, B., Jonsson, B., & Cassel, S. (2012). Inferring Canonical Register Automata. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI, Proceedings*. Springer. doi:10.1007/978-3-642-27940-9_17 (5, 42, 81, and 106)

Howar, F., Steffen, B., & Merten, M. (2011). Automata Learning with Automated Alphabet Abstraction Refinement. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI, Proceedings*. Springer. doi:10.1007/978-3-642-18275-4_19 (43 and 106)

Hungar, H., Niese, O., & Steffen, B. (2003). Domain-Specific Optimization in Automata Learning. In *Computer Aided Verification, 15th International Conference, CAV, Proceedings*. Springer. doi:10.1007/978-3-540-45069-6_31 (4 and 42)

Ip, C. N. & Dill, D. L. (1996). Better Verification Through Symmetry. *Formal Methods in System Design*, *9*(1/2), 41–75. doi:10.1007/BF00625968 (9)

Isberner, M. (2015, October). *Foundations of Active Automata Learning: An Algorithmic Perspective*. (PhD thesis). Technical University of Dortmund, Germany. Retrieved from https://eldorado.tu-dortmund.de/bitstream/2003/34282/1/Dissertation.pdf (13 and 76)

Isberner, M., Howar, F., & Steffen, B. (2013). Inferring Automata with State-Local Alphabet Abstractions. In *NASA Formal Methods, 5th International Symposium, NFM, Proceedings*. Springer. doi:10.1007/978-3-642-38088-4_9 (106)

——— (2014). Learning register automata: from languages to program structures. *Machine Learning*, *96*(1-2), 65–98. doi:10.1007/s10994-013-5419-7 (104, 141, and 144)

Jacobs, B. (2016). *Introduction to Coalgebra: Towards Mathematics of States and Observation*. (Vol. 59). Cambridge University Press. doi:10.1017/CBO9781316823187 (147)

Jacobs, B. & Silva, A. (2014). Automata Learning: A Categorical Perspective. In *Horizons of the Mind. A Tribute to Prakash Panangaden - Essays dedicated to Prakash Panangaden on the occasion of his 60th birthday*. Springer. doi:10.1007/978-3-319-06880-0_20 (87–88)

Jacobs, B., Silva, A., & Sokolova, A. (2015). Trace semantics via determinization. *J. Comput. Syst. Sci.*, *81*(5), 859–879. doi:10.1016/j.jcss.2014.12.005 (148)

Kaminski, M. & Francez, N. (1994). Finite-Memory Automata. *Theor. Comput. Sci.*, *134*(2), 329–363. doi:10.1016/0304-3975(94)90242-9 (81 and 110)

Kearns, M. J. & Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory*. MIT Press. Retrieved from https://mitpress.mit.edu/books/introduction-computational-learning-theory (3)

Kerstan, H., König, B., & Westerbaan, B. (2014). Lifting Adjunctions to Coalgebras to (Re)Discover Automata Constructions. In *CMCS, ETAPS, Revised Selected Papers*. Author. doi:10.1007/978-3-662-44124-4_10 (148)

Klin, B. & Rot, J. (2016). Coalgebraic trace semantics via forgetful logics. *Logical Methods in Computer Science*, *12*(4). doi:10.2168/LMCS-12(4:10)2016 (148)

Klin, B. & Szynwelski, M. (2016). SMT Solving for Functional Programming over Infinite Structures. In *MSFP, ETAPS*. Author. doi:10.4204/EPTCS.207.3 (77, 88, 101, 103, 110, 124, and 126–127)

Knuutila, T. (2001). Re-describing an algorithm by Hopcroft. *Theor. Comput. Sci.*, *250*(1-2), 333–363. doi:10.1016/S0304-3975(99)00150-4 (67)

Kopczyński, E. (n.d.). *Nominal LStar in LOIS*. Retrieved from https://github.com/eryxcc/lois/blob/master/tests/learning.cpp (Source code) (125)

Kopczyński, E. & Toruńczyk, S. (2016). LOIS: an Application of SMT Solvers. In *Proceedings of the 14th International Workshop on Satisfiability Modulo Theories, SMT, IJCAR*. CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-1617/paper5.pdf (107–108, 110, 124, and 126)

——— (2017). LOIS: syntax and semantics. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*. ACM. Retrieved from http://dl.acm.org/citation.cfm?id=3009876 (107–108, 110, and 124)

Kozen, D., Mamouras, K., Petrișan, D., & Silva, A. (2015). Nominal Kleene Coalgebra. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP, Proceedings, Part II*. Springer. doi:10.1007/978-3-662-47666-6_23 (107)

Krenn, W., Schlick, R., & Aichernig, B. K. (2009). Mapping UML to Labeled Transition Systems for Test-Case Generation - A Translation via Object-Oriented Action Systems. In *Formal Methods for Components and Objects - 8th International Symposium, FMCO, Revised Selected Papers*. Springer. doi:10.1007/978-3-642-17071-3_10 (52)

Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., ... Terrier, F. (2009). Papyrus UML: an open source toolset for MDA. In *Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications, ECMDA-FA*. CTIT. (53)

Lee, D. & Yannakakis, M. (1994). Testing Finite-State Machines: State Identification and Verification. *IEEE Trans. Computers*, *43*(3), 306–320. doi:10.1109/12.272431 (6, 23, 28, 31–34, 39, 43, 50, 56, and 62)

Leucker, M. (2006). Learning Meets Verification. In *Formal Methods for Components and Objects, 5th International Symposium, FMCO, Revised Lectures*. Springer. doi:10.1007/978-3-540-74792-5_6 (42)

Li, K., Groz, R., & Shahbaz, M. (2006). Integration Testing of Distributed Components Based on Learning Parameterized I/O Models. In *Formal Techniques for Networked and Distributed Systems - FORTE, 26th IFIP WG 6.1 International Conference*. Springer. doi:10.1007/11888116_31 (42)

Luo, G., Petrenko, A., & v. Bochmann, G. (1995). Selecting test sequences for partially-specified nondeterministic finite state machines. doi:10.1007/978-0-387-34883-4_6 (27)

Maler, O. & Mens, I. E. (2017). A Generic Algorithm for Learning Symbolic Automata from Membership Queries. In *Models, Algorithms, Logics and Tools - Essays dedicated to Kim Guldstrand Larsen on the occasion of his 60th birthday*. Springer. doi:10.1007/978-3-319-63121-9_8 (128)

Maler, O. & Pnueli, A. (1995). On the Learnability of Infinitary Regular Sets. *Inf. Comput.*, *118*(2), 316–326. doi:10.1006/inco.1995.1070 (90)

Mens, I. E. (2017). *Learning regular languages over large alphabets*. (PhD thesis). Grenoble Alpes University, France. Retrieved from https://tel.archives-ouvertes.fr/tel-01792635 (106)

Merten, M., Howar, F., Steffen, B., Cassel, S., & Jonsson, B. (2012). Demonstrating Learning of Register Automata. In *TACAS, ETAPS, Proceedings*. Springer. doi:10.1007/978-3-642-28756-5_32 (42)

Merten, M., Steffen, B., Howar, F., & Margaria, T. (2011). Next Generation LearnLib. In *TACAS, ETAPS, Proceedings*. Springer. doi:10.1007/978-3-642-19835-9_18 (47)

Moerman, J. (2019). Learning Product Automata. In *International Conference on Grammatical Inference, ICGI, Proceedings*. Proceedings of Machine Learning Research. (To appear) (13)

——— (n.d.). *Hybrid ADS*. Retrieved from https://github.com/Jaxan/hybrid-ads (Source code) (34)

——— (n.d.). *Nominal LStar in polynomial time?*. Retrieved from https://joshuamoerman.nl/papers/2017/17popl-learning-nominal-automata.html (Online note) (125)

——— (n.d.). *ONS Haskell Library*. Retrieved from https://github.com/Jaxan/ons-hs/ (Source code) (117)

Moerman, J. & Rot, J. (2019). *Separation and Renaming in Nominal Sets*. (Under submission) (12 and 131)

Moerman, J., Sammartino, M., Silva, A., Klin, B., & Szynwelski, M. (2017). Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*. ACM. doi:10.1145/3009837.3009879 (12, 75, 110–111, 122, and 144)

Moerman, J., Szynwelski, M., & Klin, B. (n.d.). *Nominal LStar*. Retrieved from https://github.com/Jaxan/nominal-lstar (Source code) (104)

Montanari, U. & Pistore, M. (1997). An Introduction to History Dependent Automata. *Electr. Notes Theor. Comput. Sci.*, *10*, 170–188. doi:10.1016/S1571-0661(05)80696-6 (110)

Montanari, U. & Sammartino, M. (2014). A network-conscious (π)-calculus and its coalgebraic semantics. *Theor. Comput. Sci.*, *546*, 188–224. doi:10.1016/j.tcs.2014.03.009 (76)

Moore, E. F. (1956). Gedanken–experiments on Sequential Machines. In *Sequential Machines, Automata Studies, Annals of Mathematical Studies, no.34*. Princeton University Press. (5, 26, 60, and 64)

Murawski, A. S., Ramsay, S. J., & Tzevelekos, N. (2015). Bisimilarity in Fresh-Register Automata. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*. IEEE Computer Society. doi:10.1109/LICS.2015.24 (128)

——— (2018). Polynomial-Time Equivalence Testing for Deterministic Fresh-Register Automata. In *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.MFCS.2018.72 (128)

Niese, O. (2003). *An integrated approach to testing complex systems*. (PhD thesis). Technical University of Dortmund, Germany. Retrieved from http://eldorado.uni-dortmund.de:8080/0x81d98002_0x0007b62b (76)

Object Management Group (OMG) (2004). *Unified modeling language specification: Version 2, revised final adopted specification*. Retrieved from http://www.uml.org/#UML2.0 (Website) (46 and 53)

O'Hearn, P. W. (2003). On bunched typing. *J. Funct. Program.*, *13*(4), 747–796. doi:10.1017/S0956796802004495 (132)

Peled, D. A., Vardi, M. Y., & Yannakakis, M. (2002). Black Box Checking. *Journal of Automata, Languages and Combinatorics*, *7*(2), 225–246. (4)

Petrenko, A. (1997). Technical Correspondence Comments on "A Reduced Test Suite for Protocol Conformance Testing". *ACM Trans. Softw. Eng. Methodol.*, *6*(3), 329–331. doi:10.1145/258077.265733 (26)

Petrenko, A., Li, K., Groz, R., Hossen, K., & Oriat, C. (2014). Inferring Approximated Models for Systems Engineering. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE*. IEEE Computer Society. doi:10.1109/HASE.2014.46 (21)

Petrenko, A. & Yevtushenko, N. (2014). Adaptive Testing of Nondeterministic Systems with FSM. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE*. IEEE Computer Society. doi:10.1109/HASE.2014.39 (39)

Petrenko, A., Yevtushenko, N., Lebedev, A., & Das, A. (1993). Nondeterministic State Machines in Protocol Conformance Testing. In *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems*. North-Holland. (27)

Pitts, A. M. (2013). *Nominal sets: Names and symmetry in computer science* (S. Abramsky, P. Aczel, Y. Gurevich, & J. Tucker, Eds.). Cambridge University Press. (76, 84, 87, 110–111, 132–134, 136, and 140)

——— (2014). Nominal Presentation of Cubical Sets Models of Type Theory. In *20th International Conference on Types for Proofs and Programs, TYPES*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2014.202 (135)

——— (2016). Nominal techniques. *SIGLOG News*, *3*(1), 57–72. doi:10.1145/2893582.2893594 (110 and 132)

Ploeger, B. (2005). *Analysis of concurrent state machines in embedded copier software*. (thesis). Master's thesis, Eindhoven University of Technology. (43 and 56)

Raffelt, H., Merten, M., Steffen, B., & Margaria, T. (2009). Dynamic testing via automata learning. *STTT*, *11*(4), 307–324. doi:10.1007/s10009-009-0120-7 (43)

Raffelt, H., Steffen, B., Berg, T., & Margaria, T. (2009). LearnLib: a framework for extrapolating behavioral models. *STTT*, *11*(5), 393–407. doi:10.1007/s10009-009-0111-8 (42)

Rivest, R. L. & Schapire, R. E. (1993). Inference of Finite Automata Using Homing Sequences. *Inf. Comput.*, *103*(2), 299–347. doi:10.1006/inco.1993.1021 (39 and 90)

Rot, J. C. (2015). *Enhanced Coinduction*. (PhD thesis). Leiden University, the Netherlands. Retrieved from http://hdl.handle.net/1887/35814 (36)

Rutten, J. J. M. M. (1998). Automata and Coinduction (An Exercise in Coalgebra). In *CONCUR '98: Concurrency Theory, 9th International Conference, Proceedings*. Springer. doi:10.1007/BFb0055624 (36)

_____ (2000). Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, *249*(1), 3–80. doi:10.1016/S0304-3975(00)00056-6 (147)

Sabnani, K. K. & Dahbura, A. T. (1988). A Protocol Test Generation Procedure. *Computer Networks*, *15*, 285–297. doi:10.1016/0169-7552(88)90064-5 (26)

Sakamoto, H. (1997). Learning Simple Deterministic Finite-Memory Automata. In *Algorithmic Learning Theory, 8th International Conference, ALT, Proceedings*. Springer. doi:10.1007/3-540-63577-7_58 (9 and 105)

Schöpp, U. (2006). *Names and binding in type theory*. (PhD thesis). University of Edinburgh, UK. Retrieved from http://hdl.handle.net/1842/1203 (136)

Schuts, M., Hooman, J., & Vaandrager, F. W. (2016). Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report. In *Integrated Formal Methods - 12th International Conference, IFM, Proceedings*. Springer. doi:10.1007/978-3-319-33693-0_20 (4)

Segoufin, L. (2006). Automata and Logics for Words and Trees over an Infinite Alphabet. In *Computer Science Logic, 20th International Workshop, CSL, Proceedings*. Springer. doi:10.1007/11874683_3 (110)

Selic, B., Gullekson, G., & Ward, P. T. (1994). *Real-time object-oriented modeling*. Wiley. (46)

Shinwell, M. R. (2006). Fresh O'Caml: Nominal Abstract Syntax for the Masses. *Electr. Notes Theor. Comput. Sci.*, *148*(2), 53–77. doi:10.1016/j.entcs.2005.11.040 (101 and 107–108)

Shinwell, M. R. & Pitts, A. M. (2005, feb). *Fresh objective Caml user manual*. (Technical report, No. UCAM-CL-TR-621). University of Cambridge, Computer Laboratory. Retrieved from https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-621.pdf (128)

Silva, A., Bonchi, F., Bonsangue, M. M., & Rutten, J. J. M. M. (2013). Generalizing determinization from automata to coalgebras. *Logical Methods in Computer Science*, *9*(1). doi:10.2168/LMCS-9(1:9)2013 (149)

Simão, A. & Petrenko, A. (2010). Fault Coverage-Driven Incremental Test Generation. *Comput. J.*, *53*(9), 1508–1522. doi:10.1093/comjnl/bxp073 (38)

_____ (2014). Generating Complete and Finite Test Suite for ioco: Is It Possible?. In *Proceedings Ninth Workshop on Model-Based Testing, MBT*. Author. doi:10.4204/EPTCS.141.5 (39)

Simão, A., Petrenko, A., & Yevtushenko, N. (2009). Generating Reduced Tests for FSMs with Extra States. In *TestCom/FATES, Proceedings*. Springer. doi:10.1007/978-3-642-05031-2_9 (38)

Simmons, H. (n.d.). *The topos of actions on a monoid*. Retrieved from http://www.cs.man.ac.uk/~hsimmons/DOCUMENTS/PAPERSandNOTES/Rsets.pdf (Unpublished manuscript, number 12N) (134)

Smeenk, W., Moerman, J., Vaandrager, F. W., & Jansen, D. N. (2015a). Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM, Proceedings*. Springer. doi:10.1007/978-3-319-25423-4_5 (11, 41, and 71–72)

——— (2015b). Applying Automata Learning to Embedded Control Software. Author. Retrieved from http://www.mbsd.cs.ru.nl/publications/papers/fvaan/ESM/ (Learned models and resources) (44)

Smetsers, R. & Moerman, J. (n.d.). *Partition*. Retrieved from https://github.com/Jaxan/partition (Source code) (71)

Smetsers, R., Moerman, J., & Jansen, D. N. (2016). Minimal Separating Sequences for All Pairs of States. In *Language and Automata Theory and Applications - 10th International Conference, LATA, Proceedings*. Springer. doi:10.1007/978-3-319-30000-9_14 (11 and 59)

Smetsers, R., Moerman, J., Janssen, M., & Verwer, S. (2016). Complementing Model Learning with Mutation-Based Fuzzing. *CoRR, abs/1611.02429*. Retrieved from http://arxiv.org/abs/1611.02429 (13)

Smetsers, R., Volpato, M., Vaandrager, F. W., & Verwer, S. (2014). Bigger is Not Always Better: on the Quality of Hypotheses in Active Automata Learning. In *Proceedings of the 12th International Conference on Grammatical Inference, ICGI*. JMLR.org. Retrieved from http://jmlr.org/proceedings/papers/v34/smetsers14a.html (57)

Staton, S. (2007). *Name-passing process calculi: operational models and structural operational semantics*. (PhD thesis). University of Cambridge, UK. Retrieved from https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-688.pdf (149)

Steffen, B., Howar, F., & Merten, M. (2011). Introduction to Active Automata Learning from a Practical Perspective. In *Formal Methods for Eternal Networked Software Systems - 11th International School SFM, Advanced Lectures*. Springer. doi:10.1007/978-3-642-21455-4_8 (42)

Szynwelski, M. (n.d.). *Nλ*. Retrieved from https://www.mimuw.edu.pl/~szynwelski/nlambda/ (Website & Source code) (103)

Tappler, M., Aichernig, B. K., & Bloem, R. (2017). Model-Based Testing IoT Communication via Active Automata Learning. In *ICST, Proceedings*. IEEE Computer Society. doi:10.1109/ICST.2017.32 (4)

Türker, U. C. & Yenigün, H. (2014). Hardness and inapproximability of minimizing adaptive distinguishing sequences. *Formal Methods in System Design*, *44*(3), 264–294. doi:10.1007/s10703-014-0205-0 (39)

Urban, C. & Tasson, C. (2005). Nominal Techniques in Isabelle/HOL. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Proceedings*. Springer. doi:10.1007/11532231_4 (128)

Vaandrager, F., Aarts, F., van den Bos, P., Fedotov, A., Fiterau-Brostean, P., Howar, F., ... de Ruiter, J. (n.d.). *The Automata Wiki*. Retrieved from http://automata.cs.ru.nl/ (Website) (39 and 44)

Vaandrager, F. W. (2017). Model learning. *Commun. ACM*, *60*(2), 86–95. doi:10.1145/2967606 (110 and 126)

Valmari, A. & Lehtinen, P. (2008). Efficient Minimization of DFAs with Partial Transition Functions. In *Symposium on Theoretical Aspects of Computer Science, STACS*. Author. Retrieved from http://arxiv.org/abs/0802.2826 (62)

Vasilevskii, M. P. (1973). Failure diagnosis of automata. *Cybernetics and Systems Analysis*, *9*(4), 653–665. doi:10.1007/BF01068590 (Translated from Kibernetika, No. 4, pp. 98-108, July-August, 1973) (6, 26, 35, 43, and 49)

Venhoek, D. & Moerman, J. (n.d.). *ONS*. Retrieved from https://github.com/davidv1992/ONS (Source code) (117)

Venhoek, D., Moerman, J., & Rot, J. (2018). Fast Computations on Ordered Nominal Sets. In *Theoretical Aspects of Computing - ICTAC - 15th International Colloquium, Proceedings*. Springer. doi:10.1007/978-3-030-02508-3_26 (12 and 109)

van den Bos, P., Janssen, R., & Moerman, J. (2017). n-Complete Test Suites for IOCO. In *ICTSS 2017 Proceedings*. Springer. doi:10.1007/978-3-319-67549-7_6 (13 and 39)

_____ (2018). n-Complete Test Suites for IOCO. *Software Quality Journal*. Advanced online publication. doi:10.1007/s11219-018-9422-x (13)

van den Bos, P. & Stoelinga, M. (2018). Tester versus Bug: A Generic Framework for Model-Based Testing via Games. In *Proceedings GandALF*. Open Publishing Association. doi:10.4204/EPTCS.277.9 (39)

Titles in the IPA Dissertation Series since 2016

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broștean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java*

Code. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages*. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

Curriculum Vitae

Joshua Moerman was born in 1991 in Utrecht, the Netherlands. After graduating gymnasium at the Christiaan Huygens College in Eindhoven, 2009, he followed a double bachelor programme in *mathematics* and *computer science* at the Radboud University in Nijmegen. In 2013, he obtained both bachelors *summa cum laude* and continued with a master in mathematics. He obtained the degree of Master of Science in Mathematics *summa cum laude* in 2015, with a specialisation in algebra and topology.

In February 2015, he started his Ph.D. research under supervision of Frits Vaandrager, Sebastiaan Terwijn, and Alexandra Silva. This was a joint project between the computer science institute (iCIS) and the mathematics departement (part of IMAPP) of the Radboud University. During the four years of his Ph.D. research, he spent a total of six months at the University College London, UK.

As of April 2019, Joshua works as a postdoctoral researcher in the group of Joost-Pieter Katoen at the RWTH Aachen, Germany.