Automating Proofs with State Machine Inference

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Thomas Glenn Gransden
Department of Informatics
University of Leicester

November 2016

Abstract

Automating Proofs with State Machine Inference

by Thomas GRANSDEN

Interactive theorem provers are tools that help to produce formal proofs in a semi-automatic fashion. Originally designed to verify mathematical statements, they can be potentially useful in an industrial context. Despite being endorsed by leading mathematicians and computer scientists, these tools are not widely used. This is mainly because constructing proofs requires a large amount of human effort and knowledge. Frustratingly, there is limited proof automation available in many theorem proving systems.

To address this limitation, a new technique called SEPIA (Search for Proofs Using Inferred Automata) is introduced. There are typically large libraries of completed proofs available. However, identifying useful information from these can be difficult and time-consuming. SEPIA uses state-machine inference techniques to produce descriptive models from corpora of Coq proofs. The resulting models can then be used to automatically generate proofs. Subsequently, SEPIA is also combined with other approaches to form an intelligent suite of methods (called Coq-PR^3) to help automatically generate proofs. All of the techniques presented are available as extensions for the ProofGeneral interface.

In the experimental work, the new techniques are evaluated on two large Coq datasets. They are shown to prove more theorems automatically than compared to existing proof automation. Additionally, various aspects of the discovered proofs are explored, including a comparison between the automatically generated proofs and manually created ones. Overall, the techniques are demonstrated to be a potentially useful addition to the proof development process because of their ability to automate proofs in Coq.

Acknowledgements

You would not be reading this thesis were it not for the support of my supervisors Dr Neil Walkinshaw and Professor Rajeev Raman. They have allowed me the freedom to pursue many of the ideas presented within this thesis. They have also provided many interesting and stimulating discussions on the topic. I must also thank them both for supporting (both financially and personally) my progress through the ICT Pioneers competition. Finally, I am grateful to the EPSRC and the College of Science and Engineering for funding my research through a PhD studentship.

I am also grateful to my examiners Dr Tom Ridge and Professor Alan Bundy. They made numerous comments during the viva that helper improve the overall presentation of this thesis. I also thank them for their enthusiasm for the technique and the questions asked during the viva.

Over the last few months, my work colleagues Rob, Paul, Kevin, Chris, George and Neil have offered encouragement and laughter in (roughly) equal measure. I am thankful for all of the support that has been offered, especially during my write-up. Although all of the beatings at lunchtime table tennis have been demoralising.

The next people I must thank are my in-laws - Carole, Martin, Tom and Les. Although they had no real idea what this research was about, their support and love throughout has been greatly appreciated. I think they now realise that this work has been slightly more involved than just fixing computers!

My Nana deserves more thanks than I can ever say in this thesis. Her constant support and love throughout my life has shaped the person that I am today. Despite anything that has happened to her, she always manages to put everyone before herself. I hope that I have made her proud by completing this thesis.

Last but not least, I must thank my wife Charlotte. From the moment we met she has always been supportive of everything I have done, and actively encouraged me to take this position 4 years ago. Throughout everything that has happened since we have been together, she has been right there by my side. She gives me the strength and passion to work hard, and everything I do in life I do it for her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Computer-assisted proof is a technique dating back to at least the 1960's. The process can be described as completing mathematical proofs with (at least partial) help from a computer. An interactive theorem prover (or proof assistant) is one such tool to help with this process. Given a statement expressed in the underlying logic of the tool, a proof is found via a man-machine collaboration. Producing so-called "formal proofs" generates a reproducible proof that can be verified automatically on a computer.

Many mathematical theorems have had proofs verified using interactive systems[1]. Gonthier has successfully used Coq to verify proofs of the Four-Color (Gonthier, 2008) and Feit-Thompson (Gonthier et al., 2013) theorems. Hales recently completed his multi year effort to provide a formal proof of the Kepler conjecture using Isabelle (Nipkow, Wenzel, and Paulson, 2002) and HOL Light (Harrison, 2009), whilst Mizar (Naumowicz and Korniłowicz, 2009) has the largest library of formal mathematics.

A popular application of interactive theorem proving is to prove the correctness of computer systems. Correctness is typically defined as showing that a design meets a specification. Recent advances mean it is also possible to reason directly about the underlying code. Since the infamous 1994 Pentium IV bug, Intel (and others) now use proof assistants to help verify floating point algorithms (Harrison, 2006). More recently, safety critical systems have been formally verified. Isabelle has been used to verify the correctness of the seL4 microkernel (Klein et al., 2014), whilst Coq has been used to prove the CompCert C compiler correct (Leroy, 2009).

The success of recent large-scale proof developments has shown that interactive theorem proving is a potentially valuable technique – and one that can have a significant impact in both mathematics and industry. One aspect that isn't shown in these success stories is the scale of the proof effort required to complete the proofs. For a large proof development, there are typically thousands of intermediate proofs that need to be

[1] see Wiedijk's list of 100 mathematical theorems (Wiedijk, 2008) for further information

completed, and this process can take many years. Table 1.1 gives an overview of some measurements of effort from 5 large proof efforts (Obua et al., 2014):

Development	Lines of Proof	People	Years
Flyspeck	325,000	16	2003 – 2015
seL4 kernel	200,000	17	2004 – 2009
Feit-Thompson Theorem	170,000	15	2006 – 2012
Four-Color Theorem	60,000	1	2000 – 2005
CompCert compiler	42,000	3	2005 – 2008

TABLE 1.1: Statistics from interactive proof developments

Clearly, there is a need for more automated support during proof development. The underlying logics used within proof assistants are expressive, and this impacts on the amount of automation available. Nevertheless, many approaches are available that try to automate proofs whenever possible. With the help of these approaches, each theorem prover usually has a significant library of 'exemplar' proofs available – where the correct reasoning has been applied. There are often too many proofs to study manually, however computers now have good potential for learning useful knowledge from large amounts of data.

This thesis examines the application of machine learning techniques to successful proof examples. By learning how previous proofs were completed, it is hoped that the learned knowledge can be applied to other proofs automatically. The SEPIA technique is introduced, a novel application of model inference techniques to the domain of theorem proving. Existing proofs are modelled using state machines, and the resulting models can be used to automatically generate proofs.

1.2 Contributions

The main original contributions of this thesis are:

1. The automatic translation of Coq proof scripts into a format suitable for inferring models from.

2. The application of model inference techniques to produce descriptive models of proof corpora.

3. A proof search algorithm that forms the basis of a ProofGeneral plugin that can automatically generate Coq proofs using the inferred models.

4. A suite of tools called Coq-PR^3, that form an intelligent machine learning environment for Coq.

1.3 Outline of this thesis

Chapter 2 - Background and Related Work. This chapter presents an overview of the associated literature. Firstly, the notion of interactive theorem proving is introduced, and a suitable theorem prover is selected as a basis for this thesis. Then, the area of machine learning is introduced, along with the technique of state machine inference. Finally, existing work on applying machine learning to the domain of interactive theorem proving is described.

Chapter 3 - Inferring State Machines from Proof Tactics. This chapter describes the process of modelling proof tactics. Firstly, some desirable properties of a model are considered, before state machines are selected as a formalism satisfying the criteria. Finally, the process of inferring Extended Finite State Machines from existing proof examples is shown. For some small case studies, is it demonstrated that manual inspection of the models can lead to proofs being identified.

Chapter 4 - Automating Proofs with Inferred Models. This chapter demonstrates possible ways of using the inferred models automatically. A proof search algorithm is defined that takes inferred models and produces Coq proofs. This is followed by a description of the SEPIA extension for ProofGeneral. Finally, a demonstration of the plugin shows how proofs can automatically be generated.

Chapter 5 - Extensions to SEPIA. An extension to the basic SEPIA approach is described. To achieve this, SEPIA is augmented with an additional level of machine learning using the ML4PG (Komendantskaya, Heras, and Grov, 2013) approach. The resulting suite of machine learning tools for Coq is called Coq-PR^3. The benefits of combining the two tools are described, before demonstrating how Coq-PR^3 can automatically generate Coq proofs.

Chapter 6 - Evaluation of SEPIA. This chapter evaluates the SEPIA and Coq-PR^3 approaches. Two Coq datasets are used as a basis for comparing SEPIA with other automated proof methods. A discussion of human and computer generated proofs is also included. Finally, the enhancements described in Chapter 5 are also evaluated. The results provide a number of insights into the approach, and highlight avenues for future

research.

Chapter 7 - Conclusions and Future Work. This chapter presents a summary of the work completed. Directions for future work are also considered with respect to the evaluation.

1.4 Publications

Some of the work contained in this thesis has been published at international conferences. The material that makes up the SEPIA approach (i.e. the work contained in Chapters 3 and 4) has appeared in the following two conference papers:

- Thomas Gransden, Neil Walkinshaw, and Rajeev Raman (2014). "Mining State-Based Models from Proof Corpora". In: *Intelligent Computer Mathematics*. Ed. by Stephen M. Watt et al. Vol. 8543. Lecture Notes in Computer Science. Springer, pp. 282–297

- Thomas Gransden, Neil Walkinshaw, and Rajeev Raman (2015). "SEPIA: Search for Proofs Using Inferred Automata". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, pp. 246–255

The description of the Coq-PR^3 technique (described in Chapter 5) is based on the original paper that is in preparation:

- Thomas Gransden, Ekaterina Komendantskaya, Neil Walkinshaw, Chris Warburton, and Jonathan Heras (2016). "Revisit, Reuse, Recycle your Coq Proofs: Towards an Intelligent Interactive Proof Environment". In: *Journal of Automated Reasoning*. In preparation

Various aspects of this work have also been presented at the Automated Reasoning Workshops in Dundee and Birmingham. The two short abstracts and posters are:

- Thomas Gransden (2013). "Boosting Automated Reasoning by Mining Existing Proofs". In: *20th Automated Reasoning Workshop (ARW)*

- Thomas Gransden (2015). "Combining ML4PG and SEPIA". in: *22nd Automated Reasoning Workshop (ARW)*

The work described in this thesis was also selected to be a finalist in the EPSRC ICT Pioneers Competition in 2015. The competition aims to recognize the most exceptional research in ICT based subjects. SEPIA was selected as a finalist in the Information Overload category [2].

There are also two openly-available software tools available to download:

- SEPIA is available at `https://bitbucket.org/tomgransden/sepia`
- Coq-PR[3] is available at `https://bitbucket.org/tomgransden/coqpr3`

[2] A news article about this competition can be found at `https://www.epsrc.ac.uk/newsevents/news/ukictpioneers2015winners/`

Chapter 2

Background and Related Work

2.1 Introduction

The work presented in this thesis spans the areas of theorem proving and machine learning. Therefore, an overview of these areas is necessary, along with providing any definitions required for the material presented in subsequent chapters. Firstly, the area of theorem proving is introduced, along with a description of Coq - the theorem prover that forms the basis for the work in this thesis. Then, an overview of machine learning is provided, before highlighting previous combinations of learning and theorem proving. Finally, the rest of the chapter focusses on state machine inference – the technique that the forms the basis of subsequent chapters contained in this thesis.

2.2 Interactive Theorem Proving

Interactive Theorem Provers (or proof assistants) are software tools that assist humans with the development of formal proofs. A formal proof is a logical argument that confirms the truth of some statement. The proof is built in a stepwise manner, where each small step can be verified with the help of a computer to ensure the reasoning is correct.

LISTING 2.1: A example interactive Proof

```
1  Theorem nat_add_comm : forall n m : nat, n + m = m + n.
2  Proof.
3  intros.
4  induction n.
5  simpl.
6  trivial.
7  simpl.
8  rewrite IHn.
9  trivial.
10 Qed.
```

The technique presented in the subsequent chapters focusses on the Coq theorem prover (Bertot and Castéran, 2004), a proof assistant that uses the Calculus of Inductive Constructions. To demonstrate the main ideas behind interactive theorem proving, examples using Coq will be shown. A proof begins in Coq with the user stating a proposition (see Definition 2.2.1). In Listing 2.1, the proposition has been stated in Line 1.

Definition 2.2.1. Propositions A *proposition* is an assertion expressed using logic. A proposition can be true, false or referred to as a *conjecture* – where the truth of the statement is yet to be established.

To decide if a proposition is true, a proof can be constructed and verified. Proof assistants such as Coq can be used to construct a proof. Proof assistants are built upon a small kernel – this contains a set of rules from which proofs can be constructed. Trusting the overall result means trusting the proof kernel. They are typically small so that the underlying code can be manually checked.

The proof assistant maintains a proof state (see Definition 2.2.2). When generating a proof, the user inspects the proof state, and decides how to proceed. The aim is to take the subgoal contained in the proof state and refine it into something more easily provable. This process continues until all subgoals have been proven.

Definition 2.2.2. Proof state A *proof-state* contains a context and a subgoal to prove. The context includes existing proofs, definitions and assumptions. The subgoal is the proposition that the user is trying to prove.

To aid with this process, proof assistants provide tactics (see Definition 2.2.3). There are various different commands available, depending on the action the user wishes to perform. Tactics provide an action to perform on the proof state (e.g. induction, rewrite) and usually some arguments. The arguments may refer to some already defined fact, or relate to a local assumption. In Listing 2.1, the tactics are applied in Lines 3-9.

Tactics were first introduced by Milner *et al.* (Gordon, Milner, and Wadsworth, 1979) to ease the process of finding a proof. The idea is that a tactic splits a goal into zero or more subgoals – this process repeats until no more subgoals are generated. Individual tactics can be combined using tacticals to form more complex tactics. A theorem prover can have many tactics associated with it, and many also require arguments to work properly. For instance, when performing inductive proofs the user must specify which variable to perform induction on (see Line 4 in Listing 2.1). These complications mean that finding a proof manually can still be a time consuming task.

Definition 2.2.3. Tactics A *tactic* is applied to a subgoal, to decompose it into 0 or more subgoals. Tactics are used to try and prove a proposition by using elements from the context e.g. definitions, assumptions and previously proven facts.

LISTING 2.2: Underlying CIC proof term

```
nat_add_comm =
fun n m : nat =>
nat_ind (fun n0 : nat => n0 + m = m + n0) (plus_n_O m)
(fun (n0 : nat) (IHn : n0 + m = m + n0) =>
eq_ind_r (fun n1 : nat => S n1 = m + S n0)
(plus_n_Sm m n0) IHn) n
: forall n m : nat, n + m = m + n
```

The theorem proving process involves applying tactics until no more subgoals remain. If this occurs, then a proof has been discovered and the proposition becomes a theorem (see Definition 2.2.4). In Listing 2.1, the proof is stored in Line 10 using the `Qed` command.

Definition 2.2.4. Theorems A *theorem* refers to a proposition that has a proof attached that uses mathematics and logic to establish the truth of the proposition.

Proof assistants include a pre-defined set of automated tactics. These can be used to discharge subgoals without the user having to formulate the correct commands manually. An examples of this includes the `tauto` tactic in Coq that can solve goals that are in the form a propositional tautology. This performs the necessary steps to discharge the subgoal automatically.

The user interacts with the system in terms of subgoals and tactics. Underneath this level, tactics are actually constructing proof-terms. Before the proof is accepted, the Coq kernel takes the proof term and verifies that it passes the typing rules provided by CIC. Directly producing proof terms is difficult, so the tactics allow this process to be done at a higher level that a human can understand more easily. The proof term extracted from the earlier proof example is shown in Listing 2.2.

A typical proof development involves the creation of theories (see Definition 2.2.5). Theories can contain groups of definitions and proofs about a particular concept. Theories are portable – they can be distributed to other people who can inspect the proofs and verify them by replaying the theory through the proof assistant.

Definition 2.2.5. Theory A *theory* is a collection of definitions and theorems. They are usually represented within a file, and can be imported during proof development, allowing the user to access these facts during other proofs.

Although a useful technique, interactive theorem proving is not without its drawbacks. The main limitation preventing more widespread use of proof assistants is the

lack of effective automation available. Even trivial properties will usually require human effort to identify the tactics necessary to construct a proof. The work in this thesis presents techniques that can be used to automatically formulate the necessary commands during proof attempts.

2.3 Machine Learning

This section aims to provide a general overview of machine learning - many standard textbooks are available that cover the topic in much more depth (Mitchell, 1997; Hastie, Tibshirani, and Friedman, 2009). Machine learning is using a computer to learn knowledge without being explicitly programmed to do so. The overall process of machine learning can be summarized as follows by Mitchell:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

The goal of machine learning is to build or utilise algorithms to learn from data. The process usually involves creating generalizable models that give accurate predictions, or to find patterns.

2.3.1 Basic machine learning process

The first stage of machine learning is to have a dataset of observations that will be used to learn from. A dataset can be thought of as a table, where the rows are each observation (aka measurement, data point, etc), and the columns for each observation represent the features of that observation and their values. Features can be thought of as a set of properties that can describe an observation. For instance, if the dataset contained information about the weather on a particular day, then possible features could include the amount of sunlight, whether it was raining or not, and how humid it was.

When using machine learning, a dataset is usually split into two subsets. These subsets are called the training and test datasets. Once these data subsets are created from the dataset, a predictive model or classifier is trained using the training data. The resulting models accuracy is then evaluated using the test data. Machine learning uses algorithms to automatically model and find patterns in data, usually with the goal of predicting some target output or response - the typical outputs of machine learning algorithms are discussed below.

2.3.2 Types of machine learning

Machine learning is split into two main categories that represent the type of learning that occurs - "supervised" and "unsupervised". Both techniques can be valuable and which one is chosen depends on the circumstances – what kind of problem is being solved, how much time is allotted to solving it, and what kind of data is available to learn from.

Supervised learning

In supervised learning, each observation in the dataset is associated with a class label. This represents the outcome of a particular observation. For instance, gathering weather data the possible set of class variables could be $\{sunny, rainy, cloudy\}$. The goal of supervised learning is to predict the class for an unseen example - using the knowledge learned from the training data.

To demonstrate the basics of supervised learning, consider having a set of data pertaining to some football team – e.g. Leicester City. Each row of the dataset contains information about one game in the history of Leicester City. Features include who the opponent was, when the game was played, in which stadium the game was played. In addition, there could be more specialised features about attack (e.g. number of shots on target) and defence (e.g. number of saves made).

In the case of supervised learning, the overall aim could be to use this data to predict if Leicester will win or lose against a certain team during a given game. For the sake of this example the output of the machine learning algorithm will be a yes or no answer about whether Leicester will win or not and not determine the probability of such a result occurring. Since the training data contains a history of wins and losses against certain teams at stadiums, it is possible to leverage supervised learning to create a model that is able to make a prediction when given features of a match not in the training data.

Unsupervised learning

Continuing with the football example from the previous section, the other category of learning is unsupervised learning. Suppose that the aim is to find patterns in the historic data and learn something that wasn't already known, or group the team in certain ways throughout history. To do this, an unsupervised machine learning algorithm is run against the training data that clusters (groups) the data automatically, and then performs an analysis of the generated clusters. Importantly, the data used for unsupervised learning is unlabelled – meaning there is no explicit class variable recorded (the purpose of unsupervised learning isn't to predict a particular outcome).

Using an unsupervised learning algorithm on the football match dataset, the output will be groups (clusters) of football matches. With some manual analysis, one may find that these automatically generated clusters seemingly identify certain patterns that might not have been known before running the algorithm. These may include:

- A certain player having a prolific scoring record against a particular team
- Goals being scored in a certain timeframe within a game
- Negative results being achieved against teams higher up in the league table.

One potential use of unsupervised learning is for the purposes of anomaly detection. If a certain data point isn't part of any particular cluster, there may be some underlying reason for this. However, the erroneous data point wouldn't be noticed until after running an unsupervised algorithm on the data.

2.4 Combining learning and proving

In the context of theorem proving the machine learning problem can be categorized as follows (using Mitchells definitions from Section 2.3). Assume the task T is to prove theorems automatically. The experience E will be previously proven theorems available from proof libraries and other user contributions. Finally, the performance measure P might be the improvement on the percentage of theorems proven automatically.

This section begins by describing possible ways of learning from interactive proofs. Then some existing approaches that have applied machine learning to the domain of interactive proofs are reviewed. Firstly, a collection of tools to identify useful information in proof libraries are reviewed. The next set of tools use this discovered information to guide automated provers in the search for a proof. Finally, the automatic formation of proof tactics using machine learning is discussed.

2.4.1 Hierarchical Proof Patterns

When applying learning techniques to interactive theorem proving, it is important to consider exactly what to learn from. According to Grov *et al.* an interactive proof is a complex structure that can be categorized by three distinct levels (Grov, Komendantskaya, and Bundy, 2012) – the proof tree, the subgoals and the tactic applications. Each of these possible levels may be useful for discovering patterns in proofs and generalizing the patterns to automate other proofs.

Looking at the level of tactics, a successful proof will always use a finite number of tactics, and this property makes them appealing when considering them as possible

candidates for learning. Additionally, a number of tactic combinations may bear a proof for a particular goal, and *vice versa*, different goals may be proven by the same sequence of tactics. However, tactics usually require complex arguments in order to be applied successfully. Learning from the level of tactics alone means that any information about the proof state is lost.

Another possible view of a proof is looking at the subgoal changes over the course of a proof. As discussed later in this chapter, choosing features from subgoal structures can be extremely useful for many purposes in theorem proving. Learning from this level also has limitations – any information about how the goal was produced along with the overall proof structure is lost. Additional considerations when learning from this level include whether to include context information or the goal in isolation.

Finally, the more canonical representation of a proof is to consider the proof tree. This representation shows the overall proof flow by showing relations between subgoals and tactic applications. As with the other levels, it may be the case that similarities in the proof tree can be discovered, whereas there are no evident patterns in the other levels. The limitation with this approach is that the proof trees can vary dramatically in complexity and size, and so selecting and extracting features may be difficult without pre-processing.

There is clearly a wide scope for applying machine learning to theorem proving, and there are naturally many unanswered questions that arise from such research. For instance, is there any particular level that is "best" for learning from proofs, and how to generalise after identifying patterns within a proof corpora. The rest of this section looks at some applications that learn from the different levels of proofs, and how they generalise the acquired knowledge.

2.4.2 Library search mechanisms

Interactive Theorem Provers typically have large libraries of existing proofs associated with them. The size of these libraries can easily reach thousands of lemmas and definitions, and for a user it can be difficult to know what is contained in these libraries. The Mizar Mathematical Library (Naumowicz and Kornilowicz, 2009) is one of the largest collections of formalized proofs; containing over 56000 theorems and 11000 definitions[1]. Other collections of proofs such as Isabelle's Archive of Formal Proofs also run into the thousands of facts (Blanchette et al., 2015). One set of tools to help identify facts contained within these libraries are search mechanisms.

[1]Obtained from `http://mmlquery.mizar.org/`

Identifying useful facts in a proof library

Typically searches are symbolic, meaning that the user enters some sort of pattern and the search mechanism matches against the input pattern. Search mechanisms are useful as they save the user from manually searching through proof libraries. Typical queries that these methods try to answer might be:

- Identify existing facts that contain a certain predicate.
- Search for existing facts that match a particular pattern.
- Find rewrite rules that match a specified pattern.
- Find theorems in a particular theory.
- Find theorems with a name matching the query.

As an example of how these tools work, consider the following example of symbolic fact search in Coq. The `SearchAbout` method takes as input a predicate and returns all loaded facts related to the predicate. This is done by simply looking at each theorem and definition, and checking whether the predicate is contained within it or not. In Listing 2.3, a user is searching for existing facts related to the "less than or equal to" operator (named `le` in Coq).

LISTING 2.3: Using `SearchAbout` command in Coq

```
(* Search for facts related to less than *)
Coq > SearchAbout le.

le_n: forall n : nat, n <= n
le_S: forall n m : nat, n <= m -> n <= S m
..
min_l: forall n m : nat, n <= m -> min n m = n
min_r: forall n m : nat, m <= n -> min n m = m
```

As the output in Listing 2.3 shows, the search returns a list of all known facts containing the `le` operator. It is then up to the user to decide if any of these facts are useful to them during a proof attempt. For a widely used predicate, there may be hundreds of results returned. Another use of this information is to save redundancy - if a theorem has already been defined in a library, there is no need to derive another proof of the same property.

A typical interactive proof will rely on a number of previously proven facts. Theories are built in an incremental nature, with later proofs ideally making extensive use

of earlier ones. Naturally, as the size of the development increases, the availability of existing facts also grows to a level that is hard to process manually. Symbolic search methods can only search if the patterns are known in advance. Machine learning techniques can be used to identify existing facts that *may* be useful in proving some new, arbitrary goal without knowing exactly what to look for.

The Mizar Proof Advisor is one of the earliest examples of using machine learning to identify promising facts in the Mizar Mathematical Library (Urban, 2005). The goal of the proof advisor is to answer the following query: Given a goal and a large body of knowledge, identify previously defined facts that may be useful in proving the goal. The technique uses the level of subgoals as a basis for machine learning. The desired output is a list of MML theorems that are ordered in terms of their chance of being useful in the current proof.

For each target (MML theorem), the aim is to categorize it by its features. The selected features are the predicates ("constructors" in Mizar terminology) that appear within the theorem statement, plus any other theorems/definitions that were used in the proof. This leads to an extremely large feature space: over 33,000 previously proven theorems being categorised by 40,000 references and 7,000 features. The SNoW Learning Architecture (Carlson et al., 1999) is used as it can learn efficiently in domains with large feature spaces and numerous targets.

The Mizar Proof Advisor was evaluated by Urban on 33,527 examples from the Mizar Mathematical Library (Urban, 2005). The corpus being split into 10 equally large sets. Over 10 runs, SNoW was trained on nine of the sets, with the remaining set being used to evaluate the predictions output from the system. Given each example, SNoW outputs the list of hints that have been evaluated as useful. These facts that were actually used in the MML proof are compared with the list of hints. The results produced from the proof advisor are encouraging – for an unseen goal the system will typically output 70% of the necessary facts within the first 100 suggestions.

Identifying families of similar theorems

Another application of learning from proof libraries is to use clustering techniques to produce proof families – groups of proofs that are similar based on their features. This is the approach taken in the Machine Learning for Proof General (ML4PG) system (Komendantskaya, Heras, and Grov, 2013). ML4PG is used with the Coq theorem prover, and also aims to provide statistical proof hints during the current proof attempt. However, the learning process is very different from the proof advisors described previously.

ML4PG aims to provide groups of theorems whose proof structure contains similarities. These groups are then shown to the user who can inspect the proof structure and formulate their own proof strategy from them. In the other tools such as Mizar Proof Advisor, the output is a set of existing theorems that could potentially be applied during proof attempts. Various possible uses of the ML4PG outputs are possible (Heras and Komendantskaya, 2014), such as multi-team proof development, and detecting common proof patterns in a large development.

ML4PG extracts features from Coq propositions and proofs, and can learn from all three of the levels described earlier – the proof tree, the subgoals and the tactic level. Features are captured by analysing the tactics entered by the user, and the proof state resulting from each tactic application. Interestingly, unlike the previous approaches the size of the feature vector is set at 30.

Fixing the size of feature vectors at 30 inevitably means that longer proofs cannot be captured within one feature vector. ML4PG implements the *proof patch* method – this means that features of one large proof are captured by analysing smaller proof fragments. Each feature vector can handle statistics from 5 consecutive lines of the proof. Accordingly, features from subgoals 1-5 are stored in one feature vector, 6-10 in another etc. An example of the feature extraction table is shown in Table 2.1.

TABLE 2.1: ML4PG Feature Table Example

	tactics	N tactics	arg type	tactic arg is hyp?	top symbol	n subgoals
g1	intros	1	nil	nil	forall	0
g2	induction	1	nat	Hyp	equal	2
g3	simpl	1	nil	nil	equal	0
g4	trivial	1	nil	nil	equal	0
g5	simpl	1	nil	nil	equal	0

The feature vectors are passed to clustering algorithms in either the Weka framework (Witten and Frank, 2005) or MATLAB. The output of these tools are families of theorems that have been deemed similar based on highly correlated feature vectors. The user can then look at the clusters of theorems and extract a proof strategy from the most closely correlated theorems.

One downside of the proof advisor approaches is that the resulting output is still left to the user for analysis. Being given a related family of proofs or a set of suggestions to apply is useful but only if the user knows how to interpret such information. The next set of methods in this chapter take the output from tools such as the Mizar Proof Advisor and try to use them as a basis for automated theorem proving.

2.4.3 Linking with ATP's

One line of research has been the combination of higher-order interactive provers with first-order automated provers. Proofs can be constructed automatically by providing an interface between interactive and automated provers. Although a simple idea in practice, there are many hurdles to overcome regarding the exchange of information between two very different systems. The renowned system of this kind is Sledgehammer for Isabelle HOL (Meng, Quigley, and Paulson, 2006).

Sledgehammer is a tool that can discharge interactive proof goals by harnessing the combined power of automated tools such as ATPs and SMT/SAT solvers. For a given interactive proof obligation, Sledgehammer heuristically selects a number of relevant facts (axioms, definitions, theorems) from the many thousands that may be loaded into the current session. The goal and selected facts are then converted into an ATP problem.

This means having a knowledge base (the selected facts) and some statement that is to be proven (in this case the interactive goal). The tools that Sledgehammer interfaces with can be run in parallel, all attempting to find a proof. If one of the automatic tools finds a proof, the facts used are given back to Isabelle and the proof is reconstructed within Isabelle's logic.

Due to the success of Sledgehammer, systems of this kind are now commonly referred to as "hammer" systems. Other "hammer" systems are also available – namely HOL(y)Hammer for HOL Light and MizAR for Mizar. As Blanchette *et al.* describe (Blanchette et al., 2016), hammer systems typically have (some of the following) three main features:

1. The *premise selector* (as discussed earlier) selects the most promising existing facts to use to discharge the current goal

2. The *translation module* constructs an ATP problem from the chosen premises and the current goal

3. The *reconstruction module* post-processes the ATP proof into a format that the interactive system will accept.

The translation module and reconstruction modules aren't exposed to machine learning, and so this discussion focuses on the enhancement of premise selection. In the original Sledgehammer implementation, the way premises were selected is based on the Meng-Paulson (Meng and Paulson, 2009) relevance filer (MePo for short). This was a simple approach that used symbol based matching in order to select relevant facts.

When Sledgehammer is invoked on a particular goal, MePo selects n facts based from the thousands of Isabelle theorems and definitions available. The process is iterative and begins with a set of relevant symbols (initially the higher-order constants and fixed variables from the goal). The following two steps are performed iteratively until n facts have been selected:

1. Compute each fact's score in terms of relevant and irrelevant symbols with the following calculation - $r/(r+i)$ where r is the number of relevant symbols and i the number of irrelevant symbols.

2. Select all facts with a perfect score and additionally some of the remaining top scoring facts, and add all their symbols to the set of relevant symbols.

Within the implementation of MePo, there are additional constraints that affect the scoring calculations. For instance, local facts are preferred to global ones, and rare symbols are more heavily weighted than common ones. This means that MePo is better placed to select good facts when the goal contains rare symbols as opposed to more common ones.

Despite its relatively simple relevance filtering mechanism, Sledgehammer performed well on an empirical evaluation called the Judgement Day benchmarks (Böhme and Nipkow, 2010). When originally performed, Sledgehammer proved 45% of 1240 Isabelle theorems (selected from 7 distinct Isabelle developments). More recently, with the improvements of Sledgehammer linking with SMT solvers and the enhancements of the various ATP systems, the success rate is more like 64% (Blanchette et al., 2012).

Machine learning has successfully been applied to improve the selection of relevant facts – and in turn improves the success rates of Sledgehammer further. The MaSh (Machine Learning for Sledgehammer) technique aims to improve the original MePo filter by utilising machine learning (Blanchette et al., 2016) to select the relevant facts. Additionally, a successful premise selection tool is obtained by combining MePo and MaSh.

Each Isabelle fact has an associated feature set that can be extracted. Some possible features used in MaSh are variables, constants and the theory containing the particular fact. These features are also weighted, giving preference to certain feature classes. The aim of MaSh is to predict which facts are useful for proving a particular Isabelle goal. The main source of this information can be gleaned from inspecting the proof terms. A proof term stores all of the dependencies – the facts used to obtain the proof.

MaSh is made up of three main commands – `learn`, `relearn` and `query`. MaSh can learn a new fact and its proof. It is also possible for MaSh to forget an existing proof

for a fact and relearn a new one. Finally, the `query` command takes a goal's parents, features and produces a set of hints that could be useful for proving the goal.

The learning algorithm that MaSh uses is a weighted version of Naive Bayes. MaSh computes the estimated relevance of a fact by taking into account the number of proofs in which a fact occurs. It also computes the number of proofs associated with the features of the fact. The implementation is faster than standard Naive Bayes as only the features that describe the current goal need to be considered rather than all possible features.

MaSh was evaluated on the same Judgement Day benchmark suite that the original implementation of Sledgehammer. MaSh improves the success rate to 69.8%. Even such a modest percentage gain represents a portion of proofs that no longer needed any human assistance. This demonstrates that machine learning for premise selection purposes can lead to increased automated proof success.

Other hammer systems have also been successfully implemented for other theorem provers. The HOL(y)Hammer tool for the HOL Light theorem prover recently managed to prove 39% of 14000 lemmas from the Flyspeck development (Kaliszyk and Urban, 2014). Similarly, the MizAR tool (Kaliszyk and Urban, 2013) can discharge around 40% of the theorems in the Mizar Mathematical Library completely automatically. Overall, these methods enhance the usage of automated theorem provers by doing the necessary filtering of facts when constricting an ATP problem for a given interactive proof obligation.

2.4.4 Automated tactic formation

So far, all of the approaches have used features from the lemma statements and subgoal sequences as features for learning. Instead of learning from *what* was proven, we now examine a selection of approaches that try and learn from *how* a theorem was proven. Specifically, they take examples of commands applied in existing proofs and try to produce more generalized versions of these commands that can be applied elsewhere. These techniques are the ones most closely related to the work presented in this thesis.

There exist some automated tactics and decision procedures available in some theorem provers that try to automate proofs. For classes of problems that fall into decidable fragments of logic, proofs can be discharged automatically. Examples of these include Presburger arithmetic (proved using Isabelle's `presburger` tactic) and propositional logic (proved using Coq's `tauto` tactic).

Other types of automated tactics try to combine primitive steps with the aim of finding proofs. These typically rely on databases of existing theorems being available, and work by trying to apply these existing facts. Because they only try a small number

of atomic tactics they are not very powerful, but can be used to finish off proofs by performing routine reasoning after the expert has written the bulk of the proof.

Some theorem provers such as Coq and Isabelle provide the $\mathscr{L}$tac (Delahaye, 2000) and Eisbach (Matichuk, Wenzel, and Murray, 2014) languages respectively. These are languages that allow custom tactics and proof methods to be expressed and used within proof developments. In Coq, the CompCert development makes extensive use of custom $\mathscr{L}$tac tactics. The main reason for using custom-tactics is to avoid extensive duplication within a proof development, to write the proof procedure once and invoke it automatically on other goals. However, it is still up to human experts who ultimately decide how to write a custom tactic.

The need for more powerful automated tactics to be available has led to a strand of research that tries to learn from existing tactic applications and automatically form further useful tactics. Tactics are suitable for learning due to their availability (there are many thousands of examples available) and the fact that they contain knowledge entered by an expert. There are two existing approaches that aim to learn from tactic applications with the resulting knowledge being used to automate other proofs.

Jamnik *et al*'s approach for Ωmega

The first approach (Jamnik et al., 2003), proposed by Jamnik *et al.* was used to formulate proof methods for the Ωmega proof planning system. Proof planning (Bundy, 1988) is an approach to theorem proving which uses proof methods rather than low level logical inference rules to find a proof of a given theorem. A proof method specifies a general reasoning pattern that can be used in a proof, and typically represents a combination of atomic rules of inference.

Proof planning systems search for a proof plan of a theorem which consists of applications of several methods. An object level logical proof may be generated from the execution of a successful proof plan. Proof planning is a powerful technique because it often dramatically reduces the search space, since the search is done on the level of abstract methods rather than on the level of several inference rules that make up a method. Therefore, typically, there are fewer methods than inference rules explored in the search space.

Jamnik *et al.* show how a proof planning system can learn new tactics automatically given a typically small number of well chosen examples of related proofs of theorems. This is a significant improvement, since examples exist typically in abundance, while the extraction of methods from these examples can be considered as a major bottleneck of the proof planning methodology. The idea is that the system starts with learning

simple proof methods. As the database of available proof methods grows, the system can learn more complex proof methods.

Given two (or more) examples of Ωmega tactics, Jamnik *et al* use a method to find the least general generalization of the given set of examples. The aim is to find the smallest pattern (i.e. the least number of tactics), that is also the most specific. To demonstrate the algorithm used by Jamnik *et al.*, consider having the following two sequences (representing Ωmega tactics) – e_1 = [a,a,a,a,b,c] and e_2 = [a,a,a,b,c].

Each sequence is firstly split into sublists of all possible lengths plus the rest of the list. These are referred to as pattern lists – for e_1 the first two are {[[a],[a],[a],[a],[b],[c]], [[a,a],[a,a],[b,c]]...}. Similarly for e_2 the first two are {[[a],[a],[a],[b],[c]], [[a,a], [a,b], [c]]...}. If there is any branching in the examples (i.e. a tactic creating 2 or more sugboals), then this process repeats until no branching occurs.

For every example and pattern list, the next step looks for sequential repetitions of the same elements. Exponents are used to denote the number of repetitions, which reduces the size of the patterns. For e_1 this step introduces the following exponents: {[[a]^4,[b],[c]], [[a,a]^2, [b,c]] ...}. In the case of e_2 the following exponents can be introduced: {[[a]^3,[b],[c]], [[a,a], [a,b], [c]], ...}.

The algorithm then looks for matches in the same position where the pattern is the same but differs only by the exponent. In the example sequences, there is such a match in the first position of the pattern - [a]^4 from e_1 and [a]^3 from e_2. If no matches are found, the examples can be generalised by adding a choice operator. In the case that there are matches, if the exponents are different then they can be replaced with a star operator meaning to apply the step one-or-more times. If the exponents are the same, they stay as a constant.

By performing these steps, a possible generalisation of the two example sequences is [[a]*, [b], [c]]. If more than one generalisation remains at the end of the process, then the smallest one is chosen. If there are still multiple ones, the least general one of these is selected. The approach has been demonstrated to be useful on a number of examples, where the proofs are from a similar family.

Duncan's approach for Isabelle

The second approach (Duncan, 2008) applies machine learning techniques to the Isabelle theorem prover . The first stage of the process is the extraction of relevant data from Isabelle proofs. Then, patterns within the corpora of proofs are identified based on the number of occurrences of particular tactic applications. Finally, the discovered patterns are used to form custom Isabelle tactics that can be applied to other theorems.

Given a corpora of Isabelle proofs, the first stage of Duncan's approach is to process and extract the relevant information from the proofs. To do this, Isabelle allows the proof term to be extracted from a completed Isabelle proof. This tree structure contains all of the steps used to find the proof.

Once the proof term has been obtained, an appropriate amount of abstraction must be decided upon. Proof terms contain a large amount of extra information such as variable instantiations, the other theorems applied and the direction in which they were applied. In order to understand different abstractions, the structure of an Isabelle tactic application must be defined. At the time of Duncan's work, Isabelle used a style of proof called "apply-style". Listing 2.4 shows a simple apply-style proof in Isabelle.

LISTING 2.4: An Isabelle apply-style proof

```
theorem exI: P x ==> ∃ x. P x
apply (unfold Ex_def)
apply (rule allI)
apply (rule impI)
apply (erule allE)
apply (erule mp)
apply assumption
done
```

In an apply-style step, each application typically has a method (such as rule, erule) that specifies how a particular theorem should be applied. This is then followed with the name of the rule to be applied (e.g. allI, mp). Various abstractions are now possible based on this information, for instance one abstraction could be to capture the rule name only. If this was chosen, the the proof above would be stored as [Ex_def, allI, impI, allE, mp, assumption].

However, by leaving out the method there is extra search required when actually applying these steps to a proof therefore another abstraction is to include this - [unfold Ex_def, rule allI, rule impI, erule allE, erule mp, assumption]. Other possible abstractions include the rule name and position in the proof - [beginning Ex def, beginning allI, middle impI, middle allE, end mp, end assumption]. Two abstractions that Duncan decided would work best were the rule name only, and rule name and method. If a proof contained branching, then this is removed by linearising the proofs and remembering the branching points.

The learning process then identifies commonly occurring patterns of tactics within the chosen proof library. Markov Models can calculate the probability of something happening based on previous occurrences. The type of model used is a Variable Length

Markov Model (VLMM) (Ron, Singer, and Tishby, 1996). Being able to handle variable length data is important due to differing proof lengths. A VLMM is trained on the sequences of tactics extracted in the previous step.

The model is used to assign a probability to each possible combination of two or more consecutive steps in a proof. For instance, a proof [a,b,c,d] would garner probabilities for [a,b], [a,b,c], [a,b,c,d], [b,c], [b,c,d] and [c,d]. If a probability exists for a given sequence of tactics, then the probability is updated with the frequency that the pattern occurs.

At the end of the learning process, each possible sequence of tactics has a probability assigned to it. A threshold can be chosen that disregards any tactic sequence that doesn't meet the threshold. The remaining tactic sequences are the ones that can be deemed as commonly occurring within the selected corpus of proofs.

The final step of the approach is to combine these commonly occurring tactic sequences into new tactics. All of the discovered sequences are placed into an initial population. Then, at random two sequences are chosen and considered as candidates for combining into a new tactic. There are various operators that can be used to combine pairs of patterns. For instance, assume that two patterns P1 and P2 are being considered. The possible combinations that can occur are the following:

- Macro formation - If P1 is a complete subset of P2 (or vice versa), then the smaller pattern is considered to be a macro. For example, if P1 is the smaller pattern, then it is assigned a macro ID (say *macro*1). The occurrence of P1 in P2 is replaced by *macro*1.

- Plus introduction - If P1 and P2 differ by a repeating step, then a plus operation can be introduced. For instance, if P1=[a,b,c,c,d] and P2=[a,b,c,c,c,d] then the resulting tactic will be [a,b,c+,d], where the plus operator denotes one or more applications of that step.

- $\vee$ introduction - P1 and P2 are search for a potential $\vee$ introduction. Both patterns must begin and end the same. The different middle part forms a possible choice of tactics to apply.

- $\wedge$ (re)-introduction - Finally, if none of the combinations have worked so far, a $\wedge$ introduction can be considered. All steps which resulted in branching are known from the pattern discovery stage. All steps in P1 and P2 until a branching step is reached must be the same.

If no combinations can be performed, the pair of tactic sequences are passed back and another pair is selected. The process iterates until a predetermined number of iterations have been performed. If P1 and P2 were combined in some way, they are removed from the population and the newly formed tactic takes their place.

A manual evaluation of the learned tactics showed that around 32% of the theorems in the evaluation could have at least one learned tactic applied to them. This is not to say that a complete proof could be found using the tactic. Additionally, there are additional considerations such as extra tactic arguments that weren't included in the learning process. The applicability of the tactics were also evaluated in an automated experiment.

The newly formed tactics were evaluated on a selection of Isabelle/HOL proofs by means of an automated prover augmented with the tactics. On theorems of varying complexity, the automated prover managed to prove more theorems than simply using Isabelle automated tactics. The evaluation focused on a very small selection of elementary Isabelle proofs, so the learned tactics were by their very nature simple combinations of Isabelle tactics.

The tactics did improve on the number of theorems proven automatically in Isabelle. However the time taken to prove theorems expectedly goes up, as Isabelle is trying these newly discovered tactics too. The technique excelled at proving simple theorems, but faltered as the complexity of the theorem increases. Overall, the results demonstrate that learning from tactics could be an encouraging area to pursue further.

Both Jamnik *et al.* and Duncan take the discovered proof patterns and generalize them into more widely applicable proof methods that can be used to prove theorems automatically. Although a promising strand of research, both approaches have their limitations. The tactics that Duncan discovered contain very primitive Isabelle tactics, and therefore the complexity of theorems that can be proven is low. The patterns discovered by Jamnik *et al.* are useful – however they require a good manual selection of similar proofs. This means that the approach is only useful when proving theorems that belong to the same family of proofs e.g. group theory, set theory.

2.4.5 Learning in ATP Systems

To complete this section, it makes sense to briefly mention the benefits that machine learning has brought to automated theorem proving systems. Previously, premise selection methods have been discussed that aim to narrow the search space by limiting the facts in the knowledge base. Machine learning has also helped to enhance the actual proof search procedures that are used in ATP tools. ATP systems often have many

parameters available that the user can choose prior to a proof attempt being made. A specific choice of parameters is called a search strategy.

The choice of strategy can ultimately be the difference between finding a proof or not. The challenge, given some new problem is to learn which strategy should be used. MaLeS (Kühlwein and Urban, 2015) is a tool that can help with this problem. Implemented for the E, LEO-II and Satallax provers, MaLeS is able to help prove around 8% more problems than using the prover with its default settings. In addition to simply selecting a the best potential strategy, MaLeS can provide a strategy schedule - that is a set of strategies to run along with a runtime associated with each strategy.

Another interesting area is the creation of strategies using machine learning. The Blind Strategymaker (Urban, 2013) is the BliStr tool for the E theorem prover (Schulz, 2013) that automatically formulates useful strategies for a given problem. Similar work by Bridge (Bridge, Holden, and Paulson, 2014) also studies the problem of selecting a good proof strategy for a given problem. The difference between Bridge's work and Blind Strategymaker is that Bridge selects from already known effective strategies, the BliStr tool formulates new strategies.

2.5 State Machine Inference

The SEPIA technique presented in this thesis an the extension of an existing approach called MINT (Walkinshaw, Taylor, and Derrick, 2015). MINT has previously been applied successfully to infer models of software behaviour from samples of execution traces. MINT automatically generates state machine models of software components based on examples of their behaviour. This section provides an overview of the necessary definitions and the main steps involved in inferring a state machine from example sequences.

2.5.1 Traces

Behavioural model inference techniques such as MINT require example executions of the system under inspection. These are recorded within a generic format called a trace. Traces consist of sequences of events which are made up of a label and (optionally) some variable values. The formal definition of a trace is shown in Definition 2.5.1.

Definition 2.5.1. Events and Traces. An *event* is a tuple (l, v), where l is the name of a function/method and v is a mapping from parameter variables for l to concrete values. A *trace* is a finite sequence of events, written as $((l_0, v_0), \ldots (l_n, v_n))$.

To demonstrate what an event could look like, consider a calculator program that permits the usual simple numerical operations. One possible event could be invoking the `add` method that takes 2 arguments and adds them together. A potential encoding of the event could be `(add, (num1=4, num2=4))`. According to Definition 2.5.1, the label l is `add`, and the 2 variable values are `num1=4` and `num2=4`. Usually, when there is no ambiguity, the variable names can be omitted – leaving the event as follows: `add,4,4`.

The assumption with techniques such as MINT is that interactions with the system under inspection can be characterised in terms of particular labels and variable values. If this is possible, then traces can be extracted from the system that aim to demonstrate its behaviour. In the case of the calculator program, one particular trace could be a sequence of calls from when the calculator is turned on until it is turned off again. Such an example trace is shown in Listing 2.5.

LISTING 2.5: Example trace from calculator program

```
turn_on
add,4,4
display,8
div,8,2
display,4
mult,4,10
display,40
turn_off
```

Ultimately, the choice of precisely what to record in the trace is dependent on the user and system being analysed. There are various abstractions possible depending on what exactly is being recorded. For example, the event may be a function being called, whilst the variables could be its parameters or return value. Models of more complex systems may record methods that were called along with the value of global variables at that particular point of the execution.

As with any other machine learning technique, the overall value and accuracy of the end result is dependent on the data provided as input. The selection of input data should be as complete as possible. For instance, when modelling software systems, getting a set of traces that execute the full range of permitted operations will lead to much more useful models being inferred than from an incomplete set of traces.

Another factor is the encoding used to record the traces. This includes selecting which methods to track, and which data values to record (such as inputs, return values and global variables). The choice of encoding is largely down to the system being modelled, and for what the purpose of the inferred model will be used.

2.5.2 Finite State Machines and Extended Finite State Machines

Sets of traces contain a wealth of information about a system. The control aspect of the program is provided by the sequences of events that occur, whilst the data within a system is represented by the variables associated with each event. There are many different approaches available to model such information.

In terms of analysing the underlying data state of a system, tools such as Daikon (Ernst et al., 2007) are available that can output pre- and post-conditions. To understand the sequential behaviour of a system, the archetypical model is a Finite State Machine. The definition of a Finite State Machine is provided in Definition 2.5.2.

Definition 2.5.2. Finite State Machine. A Finite State Machine (FSM) can be defined as a tuple (S, s_0, F, L, T). S is a set of states, $s_0 \in S$ is the initial state, and $F \subseteq S$ is the set of final states. L is defined as the set of labels. T is the set of transitions, where each transition takes the form (a, l, b) where $a, b \in S$ and $l \in L$.

Although valuable, an FSM only provides a view of one aspect of the system – the sequential control. Software behaviour arises from the interplay between the control and the associated data. This has led to the creation of models that can combine control and data into a single model.

One such model is an Extended Finite State Machine (Cheng and Krishnakumar, 1993). Intuitively, an EFSM adds a memory to a conventional FSM model. The transitions between states are not only associated with a label, but also with a guard that specifies conditions that must hold with respect to variables held within the memory. For instance, a transition can place constraints on which parameters can be used with a function, or only permit a function to be called when a global variable is over a certain threshold. A formal definition of an EFSM is shown in Definition 2.5.3.

Definition 2.5.3. Extended Finite State Machine. An Extended Finite State Machine (EFSM) is a tuple $(S, s_0, F, L, V, \Delta, T)$ where S, s_0, F and L are defined as in a conventional FSM. V represents the set of data states, where a single instance v represents a set of concrete variable assignments as defined in Definition 2.5.1. The data guard set Δ is the set of data guards, where each guard δ takes the form $(l, v, possible)$, where $l \in L$, $v \in V$ and $possible \in \{true, false\}$. The set of transitions T is an extension of the conventional FSM version, where transitions take the form (a, l, δ, b) where $a, b \in S$, $l \in L$ and $\delta \in \Delta$.

2.5.3 State Machine Inference Algorithms

So far, the concept of a trace and a state machine have been introduced. The missing part of the process is how the resulting state machine is inferred from a set of traces.

The underlying idea is to take the set of traces and generalise them into a descriptive model. MINT uses a process called state-merging (Walkinshaw et al., 2013; Wieczorek, 2017) to do this.

Finite State Machine Inference

To begin with, the algorithm for conventional FSM inference is described as the EFSM version builds upon this. The FSM version accepts a set of traces (as in Definition 2.5.1) however the data values aren't taken into account, as FSM's do not incorporate data. Therefore, the resulting state machine will show possible sequencing of the labels only. Various state-merging algorithms exist that are all based on the same underlying steps, shown in Algorithm 1.

Input: $Traces$
Output: An FSM consistent with $Traces$

```
1  infer (Traces) begin
2  |  (S, s0, F, L, T) ← generatePTA(Traces)
3  |  while (s1, s2) ← choosePairs((S, s0, F, L, T)) do
4  |  |  (S, F, T) ← merge(S, F, T, s1, s2)
5  |  end
6  |  return (S, s0, F, L, T)
7  end
8  merge(S, F, T, s1, s2) begin
9  |  S ← S \ {s1};
10 |  F ← F \ {s1};
11 |  T ← changeSources(s1out, s2, T);
12 |  T ← changeDestinations(s1in, s2, T);
13 |  while (s3, s4) ← findNonDeterminism(S, T) do
14 |  |  (S, F, T) ← merge(S, F, T, s3, s4);
15 |  end
16 |  return (S, F, T)
17 end
```

Algorithm 1: FSM State Merging Algorithm

Lines 1-2 The first stage of the algorithm takes a set of traces and forms the most specific possible FSM. This is called a Prefix Tree Acceptor (PTA), and is a tree-shaped state machine that accepts exactly the sequences in $Traces$. Sequences with the same prefix share the same path from the initial state in the PTA up to the point in which they diverge. Figure 2.6 shows an example PTA for the calculator program. The first and third traces share the same path from the intiial state through to the state labelled number 3, then their paths diverge as the first trace calls the `add` method, whilst the third trace calls `display`.

(a) Example Traces

```
trace
turn_on
mult   4   2
add   0   4
add   4   4
display   8
turn_off

trace
turn_on
add   1   1
display   2
turn_off

trace
turn_on
mult   1   1
add   0   1
display   1
turn_off

trace
turn_on
mult   2   0
display   0
turn_off
```

(b) Prefix Tree Acceptor

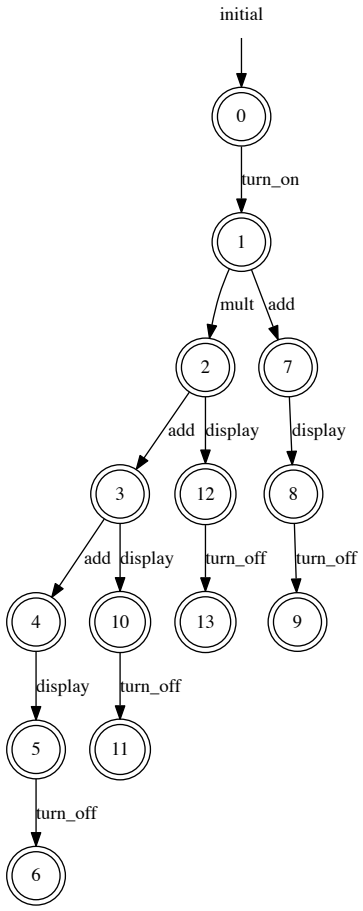


FIGURE 2.6: Set of example traces and generated PTA

Lines 3-5 The main loop of the algorithm attempts to generalize the PTA into a state-machine that fully describes the possible sequences of labels that could feasibly happen when running the system (in this case the calculator program). The challenge of state merging is to identify pairs of states that are equivalent and to merge them. Given the PTA, the process iteratively selects pairs of states and merges them. When no more equivalent pairs of states are found, the process has converged to the final state machine.

In order to deem if two states are equivalent, an equivalence score must be computed. Pairs of states are selected either naively or through heuristics such as Blue-Fringe (Lang, Pearlmutter, and Price, 1998), and their score is computed. The score takes into account the number of transitions in the outgoing paths that share the same labels. Finally, the pair of states with the highest score is suggested as the most likely to be equivalent and is returned by the `choosePairs` function.

Line 8 After a pair of states (s_1 and s_2) have been selected as merge candidates, they are provided to the `merge` function. In addition, `merge` takes the set of states S, the set of final states F and the set of transitions T. The purpose of `merge` is to modify the state machine to take into account that s_1 has been merged into s_2.

Lines 9-12 The set of states S and final states F are modified to remove s_1. Following this, the underlying transition system is modified to redirect any incoming or outgoing transitions associated with s_1. All incoming transitions into s_1 are redirected to make s_2 their destination. Any outgoing transitions from s_1 are changed to make s_2 their new source state.

Lines 13-15 Finally, the resulting transition system is checked for non-determinism and removed if encountered. This is done by recursively merging the targets of non-deterministic transitions. This process repeats until there are no more non-deterministic transitions within the state-machine. This results in an automaton that defines at most one transition for each state and each input symbol.

The main loop of Algorithm 1 repeats until there are no more pairs of states that can be merged. When this occurs, the inferred state machine is returned. For the example calculator traces, the inferred FSM is shown in Figure 2.7.

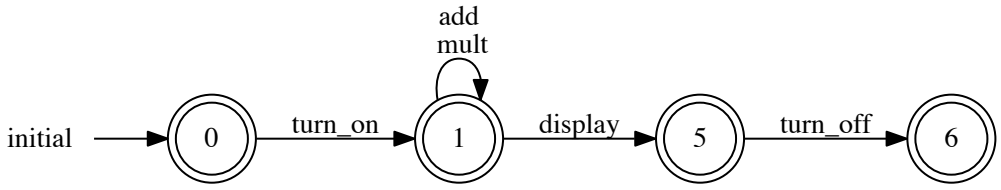


FIGURE 2.7: Inferred FSM from calculator traces

Extended State Machine Inference

The EFSM version of the algorithm (MINT) follows the same basic principles, but there are some additional steps involved. The crucial difference from before is that the data is also incorporated into the state-machine inference process. Therefore, there are some preprocessing steps that occur before the model inference process. The `infer` and `merge` functions for EFSM inference are shown in Algorithm 2.

To extend the inference approaches beyond FSMs, and to produce EFSMs it is also necessary to learn succinct rules of behaviour over the data parameters present in the traces. The process that is referred to here as "data classifier inference" encompasses a broad range of techniques that seek to identify patterns or rules between variables from a set of observations, and to map these to a particular outcome or 'class'. The possible classes could be $\{true, false\}$ if the aim is to infer whether a given set of variable values is possible or not, or more elaborate, e.g. $\{playFootball, playCricket, playTennis\}$ if the aim is to predict an activity to play based on a set of factors.

A huge variety of techniques for classifying data have been developed in the Machine Learning domain. A part of the reason for this diversity is that techniques can contain specific optimisations for their target domain. Amongst the hundreds of different techniques, core techniques include Quinlan's C4.5 Decision Trees inference algorithm (Quinlan, 1993) and Bayesian inference (Russell and Norvig, 2003) techniques such as the simple naive Bayes approach. The aim of all of these techniques is to take a set of sample observations that map a set of variable values to their respective outcome. The chosen algorithm is to produce a decision procedure that is able to correctly predict the outcome for a set of unseen variable values.

Choosing a classifier inference technique depends upon a number of factors. The choice of classifier is largely dependent on the domain in which it is being used – there is no one choice that works best in all domains. For instance one may wish to have the output in a human readable format, or there may be potential trade-offs between efficiency and accuracy. MINT uses the Weka framework of classifier inference techniques (Witten and Frank, 2005), and the default choice is decision trees – a classifier that is used exclusively throughout this thesis.

Firstly, the EFSM inference algorithm takes the original set of traces and prepares *data traces* from the originals. The data trace is grouped by label, and shows for each variable configuration what the subsequent label was. This data trace is then provided to Weka, and using the selected configuration a classifier is inferred. An example of the data trace for the calculator traces (from Figure 2.6) is shown in Listing 2.8. Using these data traces, the classifier serves to predict the next event in the trace (for example which method should be called next). An example classifier for the `add` event in the calculator program is shown in Figure 2.9.

Based on the four examples that formed the initial set of calculator traces, the classifier has learned rules that dictate the behaviour of the system (shown in Figure 2.9). For the `add` event, the classifier suggests that if the second argument to `add` is less than or equal to 1, then the subsequent event should be `display`. If the second argument is greater than 1, then the future behaviour is also dependent on the first argument.

Input: $Traces$
Output: An EFSM consistent with $Traces$

```
1  infer(Traces, k) begin
2      Failed ← ∅;
3      dataTraces ← prepareDataTraces(Traces);
4      C ← inferClassifiers(dataTraces);
5      (A, Vars) ← generatePTA(Traces, C);
6      foreach (s1, s2) ∈ choosePairs(A, C, k) \ Failed do
7          (A′, Vars′) ← merge(A, (s1, s2), Vars, C);
8          if consistent(A′, C, Vars′) then
9              A ← A′;
10             Vars ← Vars′
11         else
12             Failed ← Failed ∪ (s1, s2)
13         end
14     end
15     return A
16 end
17 merge(A, s1, s2, Vars, C) begin
18     A_S ← A_S \ {s1};
19     A_F ← A_F \ {s1};
20     A_T ← changeSources(A_T, s1_out, s2);
21     A_T ← changeDestinations(A_T, s1_in, s2);
22     while (t1, t2) ← equivalentTransitions(A_T, s2, Vars, C) ) do
23         if t1_dest == t2_dest then
24             Vars(t2) ← Vars(t2) ∪ Vars(t1);
25             A_T ← A_T \ {t1}
26         else
27             (A, Vars) ← merge(A, (t1_dest, t2_dest), Vars, C);
28         end
29     end
30     return (A, Vars)
31 end
```

Algorithm 2: EFSM State Merging Algorithm

LISTING 2.8: Data traces for calculator program

```
turn_on class=mult
turn_on class=add
turn_on class=mult
turn_on class=mult
============================
mult num1=4 num2=2 class=add
mult num1=1 num2=2 class=add
mult num1=2 num2=0 class=display
============================
add num1=0 num2=4 class=add
add num1=4 num2=4 class=display
add num1=1 num2=1 class=display
add num1=0 num2=1 class=display
============================
display num=8 class=turn_off
display num=2 class=turn_off
display num=1 class=turn_off
display num=0 class=turn_off
============================
turn_off class=n/a
turn_off class=n/a
turn_off class=n/a
turn_off class=n/a
```

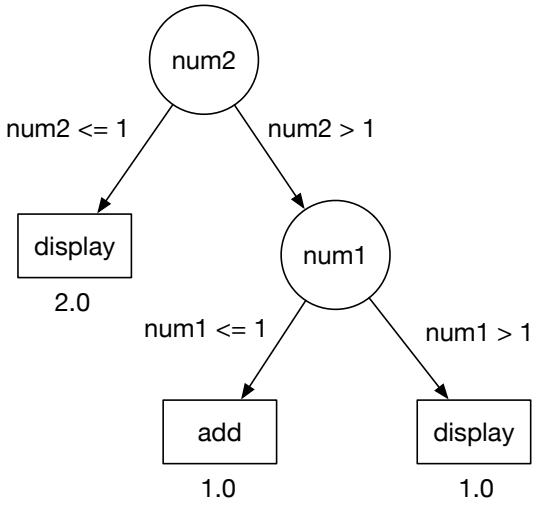


FIGURE 2.9: Graphical representation of classifier for `add`

Underneath each leaf in Figure 2.9 is a number. These numbers represent the total number of paths to that leaf divided by the number of misclassified instances. In the classifier for `add`, there are no misclassifications. Looking at the training data, there are 2 instances where the second argument to `add` is less than or equal to 1. In both these cases, the subsequent event is `display` - therefore there are no misclassified instances – the rest of the leaves also contain no misclassification.

These data-traces and generated classifiers are then used by the EFSM inference algorithm. By incorporating this information, the transitions within the state-machine contain constraints based on the classifiers. The rest of this section provides an overview of the main stages of Algorithm 2.

Lines 1-5 The initial set of traces is taken and arranged into a Prefix Tree Acceptor as before. In this PTA however, transitions are labelled not only with the name of a function, but with the data variables values that correspond to the trace element. This means that a pair of states (a,b) only share a prefix in the PTA if the inferred classifiers yield identical predictions for each data-configuration in the prefix of a as they do in b. The PTA and data classifiers are then used as a basis for state merging (similar to Algorithm 1).

Line 6 The `choosePairs` function takes more parameters into account than in the FSM inference case. k refers to an optional "minimum score" that must be achieved before a pair of states can be deemed equivalent. This score pertains to the length of the outgoing paths from both states. `choosePairs` also takes the inferred classifiers as a parameter as these play a role in computing the score of state-pairs. Because data is incorporated into the model a pair of states in an EFSM can only be equivalent if their attached data values lead to the same predictions by the classifiers.

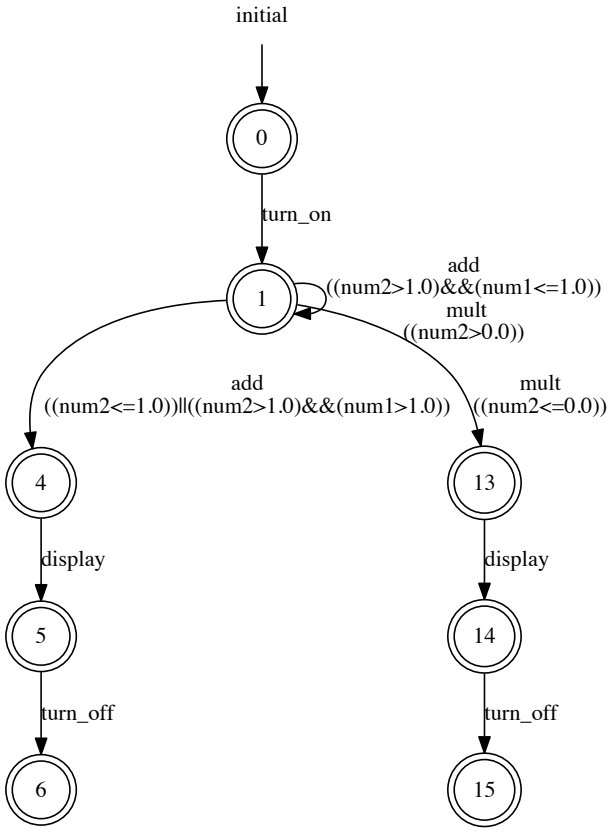


FIGURE 2.10: Inferred EFSM from calculator traces

Line 7 The `merge` function is similar to the FSM version, however it differs in the way it detects non-determinism. Instead of identifying non-determinism by looking at the transition labels, it also checks that the data variables are treated as equal by the classifiers. When transitions are merged, their sets of data values are merged as well.

Lines 8-15 After a merge has been processed, the resulting model (A') is checked by the `consistent` function. This ensures that, for each transition within the merged machine A', the attached data variables $Vars'$ are consistent with the classifiers. If `consistent` returns true, then the merge is accepted and the algorithm continues. If not, the current merge is ignored and the next merge is attempted.

The EFSM inferred from the calculator traces is shown in Figure 2.10. The underlying data classifiers lead to a much more descriptive model being inferred than before. The EFSM transitions contain not only a label, but also constraints on the underlying data state. Although in this chapter the example has been simple, in practice these descriptive models have been shown to be invaluable in the software engineering domain. As later chapters will show, these models can also be used effectively for theorem proving purposes.

2.6 Conclusions

Interactive theorem proving has been shown to be a potentially valuable technique for verifying correctness of complex systems, and for ascertaining the truth of significant mathematical statements. However, the wider usage of these tools remain limited. The main reason for this is the steep learning curve required to guide the theorem prover by specifying the required steps. There is also a varying amount of proof automation available – meaning that the human expert is left to do the core work in finding the proof.

Tool support is available for many purposes including library maintenance through to proof automation. However, the level of automation available differs on a per theorem prover basis as the logics and environments differ greatly between different systems. The Coq theorem prover is one of the most popular tools, but suffers from little effective proof automation when compared to similar tools such as Isabelle. Therefore, Coq is a sensible choice as the focus of this thesis.

One promising strand of research in computer science is machine learning. This is the process of extracting useful information from the large amounts of data that is readily available. In the theorem proving domain, there exist large corpora of proof examples where the successful steps have been input by an expert. An interactive proof is a highly structured piece of data (as summarised by Grov, Komendantskaya and Bundy). Three potential levels of a proof are the commands used when finding the proof, the subgoals generated during the proof, and the overall proof tree.

Recently, many researchers have studied applying machine learning to increase the amount of proof automation in theorem proving. The theorem provers that have benefited most from these methods are Mizar, Isabelle and HOL Light. The most successful of these approaches have concentrated on learning from goal structures and proof trees. A more unexplored area is the application of machine learning to tactic sequences – this is the area that this thesis is concerned with.

A useful technique in the domain of software engineering is the process of state machine inference. Given sequential data (e.g. executions from a software system), state machines can be used to model the overall behaviour in terms of the sequencing of possible events. Richer models can also take into account an underlying data state, meaning that extremely descriptive models can be inferred.

Existing work has studied applying learning techniques to the tactic applications within a proof (Duncan, 2008; Jamnik et al., 2003). These techniques have been shown to be useful in situations where proofs follow the same basic reasoning pattern. During the learning phase, the existing techniques abstract away information from the tactic applications such as the parameters given to tactics. The work presented in this thesis

looks to build upon these methods to improve proof automation in Coq. To achieve this, state-machine inference techniques are applied to Coq proof corpora to model the tactic applications.

Chapter 3

Inferring State Machines from Proof Tactics

3.1 Introduction

This chapter introduces the first stage of the SEPIA (Searching for Proofs using Inferred Automata) approach. Applying techniques directly to the proof tactics used to construct a proof is one of the less explored combinations of learning and proving. Many existing approaches require the proofs to be replayed through the theorem prover in order to obtain the features used for machine learning.

Interactive proofs are completed by identifying a sequence of tactics that prove a theorem. The user will have carefully made the selection of tactics and their associated arguments to complete a proof. With varying amount of proof automation available, this can become a difficult task. Even though there are large libraries of existing proofs available, manually processing these to identify proof strategies is time-consuming. SEPIA tries to address this problem by automatically producing models that provide an overview of the tactic applications within a corpus of proofs.

The work completed in this chapter shows how Coq tactic sequences can be inferred as state machines. This process happens without needing to interact with the theorem prover. The resulting models can be interpreted as showing the reasoning behaviour present in a given corpora or proofs. The models can be used to manually drive proof attempts by suggesting to the user which tactics and arguments to apply. Some examples show the potential benefit of using state machines for proof generation in Coq. This chapter described work that was presented at CICM 2014 (Gransden, Walkinshaw, and Raman, 2014).

3.2 Inferring models from Coq proofs

In Chapter 2, MINT was introduced as a way of inferring models of complex software systems. In this chapter, the application and extension of MINT to learn from Coq proofs is described. The inferred model provides a descriptive overview of the main reasoning that was used in the corpora. It can be used to understand the reasoning patterns present within a proof corpora, and be used as a basis for manual proof attempts in Coq.

Coq was selected as the main focus of this thesis for a number of reasons:

1. Popularity - Coq is a system with a large userbase, and is arguably one of the most widely used proof assistants. Additionally, it was the recipient of the ACM Software Systems Award in 2013[1].

2. Availability of proofs - Coq has a large (over 10,000 theorems) standard library[2] and various user-contributions. They are all openly available, and importantly the tactics used in the proofs are easily accessible.

3. Proof style - Coq uses a procedural proof style that is simply a collection of tactic applications. This makes learning and extracting the used proof steps simpler than using other theorem provers that employ the declarative style of proof.

4. Amount of automation - Coq has less automation than some of its contemporaries such as Isabelle, Mizar and HOL Light. Therefore, it is a good candidate to try and improve this situation.

For MINT to be successfully applied to Coq proofs, there are some requirements that will need to be satisfied:

- **Capture tactic sequences.** The model should propose sequences of tactics. These models will then contain knowledge that can be applied during other proof attempts. Traditional Finite State Machines (Definition 2.5.2) are a candidate to capture the sequential aspect of the proofs.

- **Encode tactic arguments.** Coq tactics are made up of an action (e.g. `induction`, `rewrite` and (usually) some arguments. The arguments may represent other facts to apply, variable instantiations or assumptions. Any model of tactics should ideally capture this information in order to be applicable to other proofs. The Extended Finite State Machine (Definition 2.5.3) can handle complex domains that require parameters.

[1] http://www.sigplan.org/Awards/Software/

[2] `https://coq.inria.fr/library/`

LISTING 3.1: Progress of Coq automated tactics

```
1 subgoal

a : nat
============================
a * O * S O = O
```

- **Ensure scalability.** The model should be scalable so that (in practice) any number of proofs can be used as input. There are situations where SEPIA will be used to learn from proof corpora of varying sizes. Therefore the approach should be able to infer models from any number of examples, if required.

- **Be automatically searchable.** The ultimate aim of this work is to automate the proof process. To do this, the selected model is must be stored in a format that can be automatically used during proof attempts. It must also be possible to use this information within the Coq proof environment.

This chapter focuses on the first two requirements, whilst Chapter 4 will address the remaining ones. The first stage of the SEPIA approach is to take existing Coq proof scripts and convert them into a format that the state-machine inference tools can use. Then, MINT is used to infer sequential models from the proof corpora. Finally the models are used manually to demonstrate the possibility of generating proofs using the information contained in the models.

3.2.1 Motivating Example

To motivate SEPIA, let us consider a typical scenario that arises during interactive proof development. Suppose that someone is trying to prove the following conjecture in Coq `forall a: nat, a * O * S O = O`. The user is stuck and wondering how to make progress. They first invoke Coq's collection of automated tactics.

However, these are limited in their proving capabilities and typically used for simplification. As expected, the automated tactics available in Coq fail to prove the theorem, only being able to progress to the state shown in Listing 3.1.

The user is aware of a Coq library called `ListNat` that contains 70 proofs about similar properties. Another possible approach could be to spend time manually analysing the `ListNat` library to try and elicit a proof strategy. This is a time consuming approach, and in many cases doesn't lead to a proof being discovered.

An alternative approach could be to use SEPIA to infer a model from the proofs in `ListNat`. Then, the resulting model could be used to try to generate a proof. Although

still manual work, the inferred model presents plausible proof strategies that are present in the `ListNat` corpora that could be applicable to other theorems. This chapter investigates if it is possible to infer models from a proof corpus such as `ListNat` and use these models to prove other properties not in the corpus.

3.2.2 Proof Trace Generation

A typical tactical proof script[3] contains many theorems, along with the sequence of proof steps that the expert user entered to complete the proof. Each proof step has the basic structure: *proof_method params* where *proof_method* refers to a Coq command (e.g. `rewrite, apply, intros`) and *params* constitutes the (optional) parameters provided to the Coq command. These parameters may refer to many different entities such as existing lemmas, rewrite rules or may be related to variables in the goal.

MINT requires a text file containing one or more traces. In the context of software engineering, an individual trace corresponds to a sample execution of a system. The format is sequential by nature as they contain the functions called during the execution of the program. When analysing programs, there exist tools that can automatically produce traces in the required format. However, in the case of interactive proofs, a process is required to convert the Coq proof scripts into a sequential trace format. For us, a proof forms one trace, where the trace contains the sequence of tactics (and arguments) used. A proof trace can be defined as follows (based on Definition 2.5.1):

Definition 3.2.1. Proof trace. An *event* is a tuple (t, p), where t is the name of a Coq tactic and p is a string of parameters given to the tactic. A *proof trace* is a finite sequence of events, written as $((t_0, p_0), \ldots (t_n, p_n))$.

As shown in Table 3.1, the encoding of Coq proofs is a straightforward translation. With respect to the tuple of tactics and parameters (t, p), the *proof_method* would correspond to t whilst the parameters *params* are assigned to p. Importantly, anything following the *proof_method* is considered to be included in the parameters. So each *proof_method* either has no parameters or a string containing anything that followed it. If a proof method doesn't have any parameters provided to it, this is indicated by p being left as an empty string.

Coq allows individual tactics to be chained together using a ; operator. Whereas individual tactics are always applied to the current subgoal only, the semicolon operator allows a different way of applying tactics. Consider having two Coq tactics – $\mathtt{t}_1$ and $\mathtt{t}_2$. The combined tactics $\mathtt{t}_1;\mathtt{t}_2$ has the effect of running $\mathtt{t}_1$ on the current subgoal and then running $\mathtt{t}_2$ on each generated subgoal. The semicolon is considered one of

[3] Although this work concentrates on Coq, the method in principle can be applied to other ITPs.

TABLE 3.1: Original proof and proof trace for an example lemma

(a) Proof Script

```
Lemma ex : (n*m = 0)->(n=0)\/(m=0).
intros.
induction n.
tauto.
simpl in H.
right.
assert (m <= 0); try omega.
rewrite <- H.
auto with arith.
Qed.
```

(b) Trace

Event e	Label l	Parameters p
e_0	intros	$\langle p =$ “”$\rangle$
e_1	induction	$\langle p =$ “n”$\rangle$
e_2	tauto	$\langle p =$ “”$\rangle$
e_3	simpl	$\langle p =$ “$in\ H$”$\rangle$
e_4	right	$\langle p =$ “”$\rangle$
e_5	assert	$\langle p =$ “$m \leq 0;$”$\rangle$
e_6	try	$\langle p =$ “$omega$”$\rangle$
e_7	rewrite	$\langle p =$ “$\leftarrow H$”$\rangle$
e_8	auto	$\langle p =$ “$with\ arith$”$\rangle$

the most fundamental building blocks of effective proof automation (Chlipala, 2011), resulting in more readable proof scripts.

SEPIA must take into account when tactics have been combined using the semicolon operator. The obvious choice here is to split a combined tactic into its individual components. Importantly, the semicolon usage must be recorded within the trace. As Table 3.1 shows, line 7 of the proof contains two tactics separated using the semicolon. In the trace, e_5 and e_6 capture these tactics. The semicolon is added at the end of the parameter string to denote that this particular tactic should be considered part of a combination. This information is obeyed by the proof search algorithm described in Chapter 4.

The concrete implementation details of the proof trace generation is described later in this chapter. For now, it is enough to understand that it is possible to convert Coq proof scripts into a format used by state-machine inference tools. The next focus is on how these traces are used to infer models, and how such models can be used during proof attempts.

3.2.3 Using proof traces in model inference

This section shows how the MINT technique can be used to derive sequential models from proofs. These descriptive models not only describe the possible sequences of proof steps, but also the necessary parameter values associated with these proof steps. Although previous work on inferring FSM's and EFSMs has focussed on program execution traces, they also appear to be well suited to the domain of interactive proofs where we want to capture the interplay between control (tactics) and data (parameters).

Having converted one or more Coq theories into the necessary trace format, MINT can be used to generate a model. To begin with data classifiers are inferred that, for each

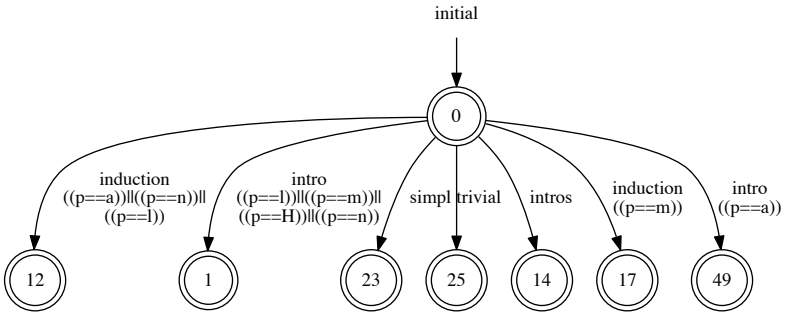


FIGURE 3.2: Initial fragment of PTA inferred from `ListNat`

proof_method, produce a function that uses the parameters to predict the subsequent tactic to be applied. An example data classifier can be seen in Figure 3.3(a) for the `induction` proof method (generated from `ListNat`).

The data classifier is interpreted as follows: if the parameters *params* are equal to *n,a* or *l*, then the subsequent proof method to be applied should be `simpl`. If *params* is equal to *m* then the following proof method should be `trivial`. The data models produced are reasonably simple because of the chosen trace encoding.

Once the data classifiers have been generated, the set of proof traces is arranged as a prefix tree. The prefix tree itself is a valid and usable state machine – however it is not practical to use because it only permits *exactly* the tactic sequences contained in the traces. The tree for this example[4] is shown in Figure 3.3(b).

To demonstrate what the prefix-tree (see Chapter 2) looks like when modelling Coq proofs, Figure 3.2 shows an enlarged initial fragment of the PTA generated from the `ListNat` traces. Recall from Chapter 2 that in a prefix tree with data, paths share the same prefix if their attached data values lead to the same prediction being suggested by the classifiers. This can be seen when considering paths that contain the `induction` tactic.

In the situations where `a`, `n` or `m` are used as parameters for `induction`, the subsequent tactic to apply is determined to be `simpl`. In the remaining case where the parameter is `m`, the next tactic is `trivial`. This behaviour can also be observed as expected in the PTA. The transition from state 0 to 12 shows that the three arguments that lead to a `simpl` tactic all share the same path in the PTA. Conversely, the transition from state 0 to 17 contains the other `induction` trace. It should be pointed out that the numbers annotating each state in the EFSM are merely there to facilitate descriptions.

Each transition in the prefix tree is associated with a tactic with the parameter values that correspond to that transition. The inference challenge for the merging algorithm is to select compatible pairs of states to be merged. These states should have similar

[4]The labels are unreadable, but the purpose is merely to give an intuition of what the tree might look like, and to illustrate the ensuing state merging challenge.

outgoing paths, and should not entail the merging of states that are incompatible and should not raise contradictions with the inferred data classifiers (as discussed in Section 2). MINT accomplishes this using the EFSM algorithm from Chapter 2.

The final EFSM is shown in Figure 3.3(c). The constraints on the transitions detail the parameter configurations that are associated with each transition. The model is deterministic; for any state there is never more than one outgoing transition for a given combination of label and variable configuration. Unlike other EFSM inference techniques, MINT uses the data as part of the inference process rather than for annotation. This helps to keep the models concise.

3.3 Qualitative comparison of inferred models

MINT is capable of inferring different types of model (FSMs or EFSMs respectively) depending on whether it is configured to use data or not. This section highlights their benefits and limitations when applied to a corpus of Coq proofs. For familiarity the `ListNat` library is used again. First a conventional FSM is inferred (i.e. a model without data), before an EFSM is inferred from the same proofs.

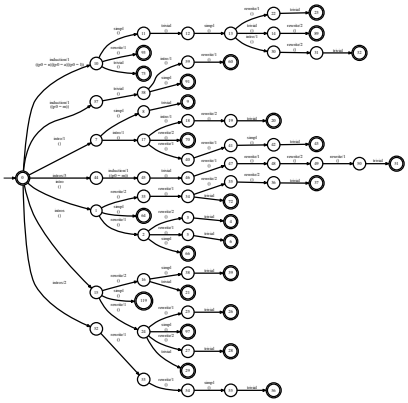
3.3.1 Inferring an FSM from `ListNat`

When only considering the tactics applied (without considering the tactic arguments), the standard FSM algorithm in MINT is used. The algorithm is essentially the same as the ESFM version, however no data classifiers are inferred, so the inference algorithm ignores the tactic parameters. The resulting FSM can be seen in Figure 3.4. The key strength of this model is its conciseness. The entire contents of the `ListNat` traces (all 70 of them) can be compacted into a state machine containing 3 states and 12 transitions.

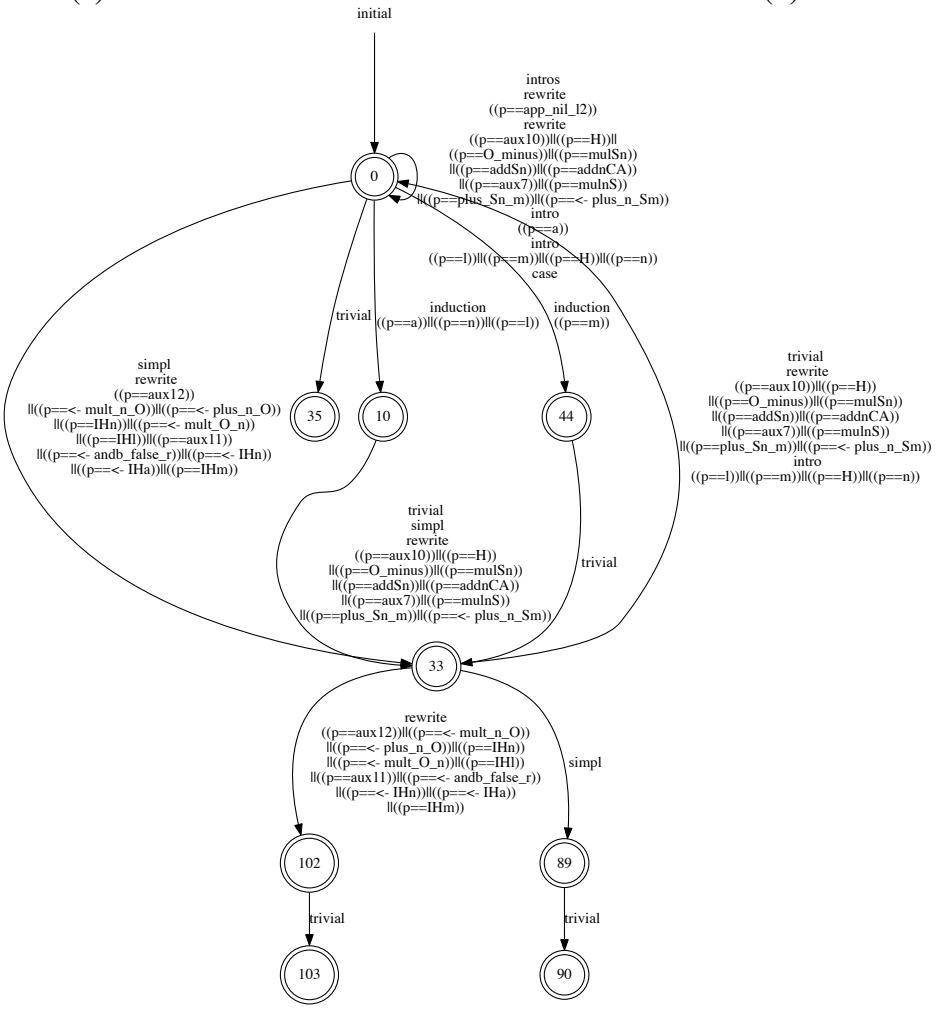
Though concise, there are downsides to inferring FSMs from Coq proofs. The model only provides a partial overview of the reasoning behaviour present within the proofs. For instance, under what circumstances should a particular tactic be applied? There is also no guidance on what parameters to provide to the tactics.

Nevertheless, for providing a quick overview of the possible sequencing of tactics when constructing a proof, an FSM can be useful. However, for the purposes of actually completing proofs using the inferred models, a more descriptive model containing the tactic arguments is needed. Otherwise, it is left to the user to guess the correct arguments – someone unfamiliar with Coq may not have this knowledge. For this reason, we consider the usage of MINT to infer EFSM's from proofs instead of conventional FSMs.

```
MODEL FOR:induction
J48 pruned tree
------------------
(p = n): simpl
(p = a): simpl
(p = m): trivial
(p = l): simpl
```



(a) Data rules for `induction` (b) Prefix tree



(c) Inferred EFSM

FIGURE 3.3: PTA and inferred EFSM for `ListNat` traces.

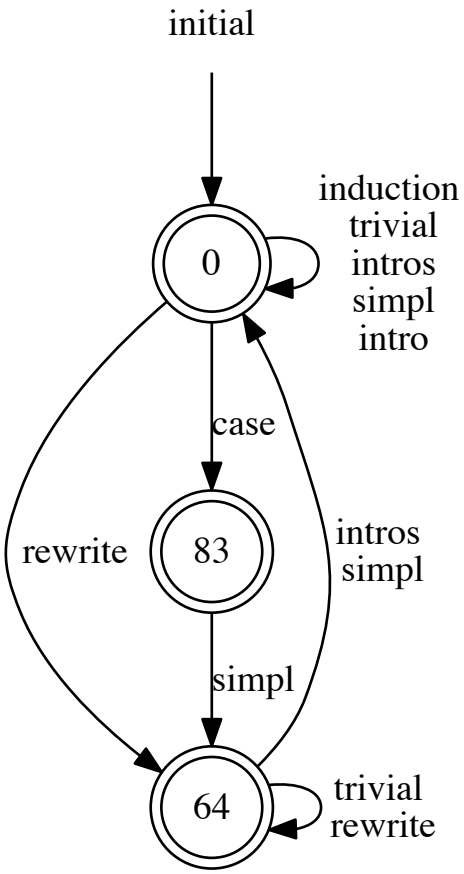


FIGURE 3.4: FSM inferred from `ListNat`

3.3.2 Inferring an EFSM from `ListNat`

As shown in Figure 3.3(c), the EFSM inferred from `ListNat` is slightly larger. Though larger, the model is still reasonably compact considering there are 70 proofs. A more diverse selection of proofs would probably lead to a larger model still.

The value of the EFSM is that it makes explicit the circumstances under which certain paths through the model should be followed. The machine contains several examples that demonstrate how the same tactic with different parameters can lead to different paths through the model. This information is not present in the standard FSM.

For the purposes of trying to derive proofs from the inferred model, the EFSM appears to be a suitable choice. Full Coq commands can be derived from the model by selecting a tactic, then an appropriate parameter as suggested on the transitions within the model. The command can then be applied to Coq, and transitions from the subsequent state can be analysed and applied. As will be shown later, the models can be used manually to derive proofs (and Chapter 4 will attempt to automate this process). The rest of this thesis focusses exclusively on using EFSM's for the purposes of proof development.

3.4 Manual application of models

To demonstrate the potential for using EFSM's to develop proofs in Coq, some examples are presented. Firstly, the motivating example from the beginning of this chapter is revisited. Then, a second small example is considered.

3.4.1 Example 1: `ListNat`

We revisit the motivating example, where the user was trying to prove the following:

```
forall a:nat, a * 0 * S (0) = 0.
```

They have already tried the automated tactics, which in this case didn't help to prove this statement. Instead, they chose to infer an EFSM from the `ListNat` theory. Figure 3.3(c) shows the EFSM inferred from the proofs in `ListNat`.

It is possible to trace a path through the inferred model manually. To accomplish this, the user steps through the machine, considering for each state the constraints that are attached to each transition. Each transition can be converted into Coq commands and input into Coq. If the proof step was successfully applied, then that particular path in the model can be explored further.

For the example, one possible path through the model to prove the lemma would be the following sequence of proof steps:

```
intro a.  rewrite <- mult_n_O. trivial.
```

By running the proof steps above in Coq, the model inferred by SEPIA has yielded the proof steps required to prove the theorem. It is important to highlight that not all paths within a model represent a proof. For any given theorem, a majority of suggested tactics may not be applicable. Instead, the model represents a search space, where the search objective amounts to finding a suitable path through the model.
As a reference, the theorem was proven by a human using the following sequence of tactics:

```
intros.  rewrite <- mult_n_O. rewrite <- mult_O_n.  trivial.
```

Interestingly, the proof found by using the EFSM was one step shorter by omitting the second `rewrite` tactic. Additionally, the sequence found from traversing the EFSM was not (at least not in its entirety) amongst the proofs in `ListNat`. The tactic sequence was only found as a result of inferring an EFSM. This is one major benefit of inferring models from proofs – the process of generating the model can identify previously unseen links between tactics that might not be spotted manually. These new tactic sequences play an important role in automating proofs. This idea is studied further in the evaluation (Chapter 6).

3.4.2 Example 2: `Le` and `Lt`

As another example, consider the following example that could arise during the proof process. A user is trying to prove the following theorem in Coq:

```
forall n m p:nat, p + n <= p + m -> n <= m
```

As usual, the automated tactics within Coq are invoked. Unfortunately, they are unable to prove the example theorem, and can only advance the proof to the state shown in Listing 3.5:

LISTING 3.5: Progress of Coq automated tactics

```
1 subgoal

n : nat
m : nat
p : nat
H : p + n <= p + m
============================
n <= m
```

For this example, the user selects two theories from the Coq standard library called `Le.v` and `Lt.v` that prove elementary properties of the $\leq$ and $<$ operators. In total, there are 40 proofs contained within these theories. SEPIA is used to infer a model from the proofs in `Le.v` and `Lt.v`. This provides a model containing possible combinations of tactics that may be useful in other proofs. There are 23 states and 37 edges within the model (shown in Figure 3.6). The model produced by SEPIA can be taken and used manually in order to find a proof.

A possible path through the EFSM can be traced and the example theorem is proven using the tactics `intros m n diff. elim diff; auto with arith.` The discovered proof is particularly interesting for two reasons. Firstly, SEPIA has proposed a sequence of proof steps that has managed to prove something that Coq's automated tools could not. Secondly, the sequence of tactics (and parameters) is new; it is not contained within `Le.v` or `Lt.v`.

Finding the proof was not necessarily a simple process. Given the model, it required a manual analysis of the outgoing tactics from each state. The user is left to use their own intuition and knowledge, using the EFSM to guide them towards a proof. Although difficult, this will be addressed by adding in automation later in Chapter 4.

3.5 Implementation

This section provides an overview of how SEPIA converts Coq proofs into a format that MINT can use for model inference. There are no existing methods available to extract tactic sequences from Coq proofs. Therefore, new functionality has been created that takes Coq proof scripts and converts them into proof traces. This process is done automatically and generates a text file containing the proof traces.

For each theory provided, the first step is to simply extract the proofs and disregard anything else such as definitions or comments. Understanding the basic structure of Coq proof scripts is essential attempting to parse them. Gallina, the input language of Coq (Bertot and Castéran, 2004) defines an extensive grammar to ensure proof scripts are constructed correctly. The most relevant parts of the grammar for parsing proof scripts is in Figure 3.7:

A valid sentence within the Gallina language (Bertot and Castéran, 2004) contains an assertion followed by its corresponding proof that has been constructed. Proofs in Coq are started with the `Lemma` or `Theorem` commands, followed by an identifier, and finally a term (the theorem statement). For the purposes of proof trace generation, none of this information is used within the trace. In Listing 3.8 this information corresponds to the first two lines.

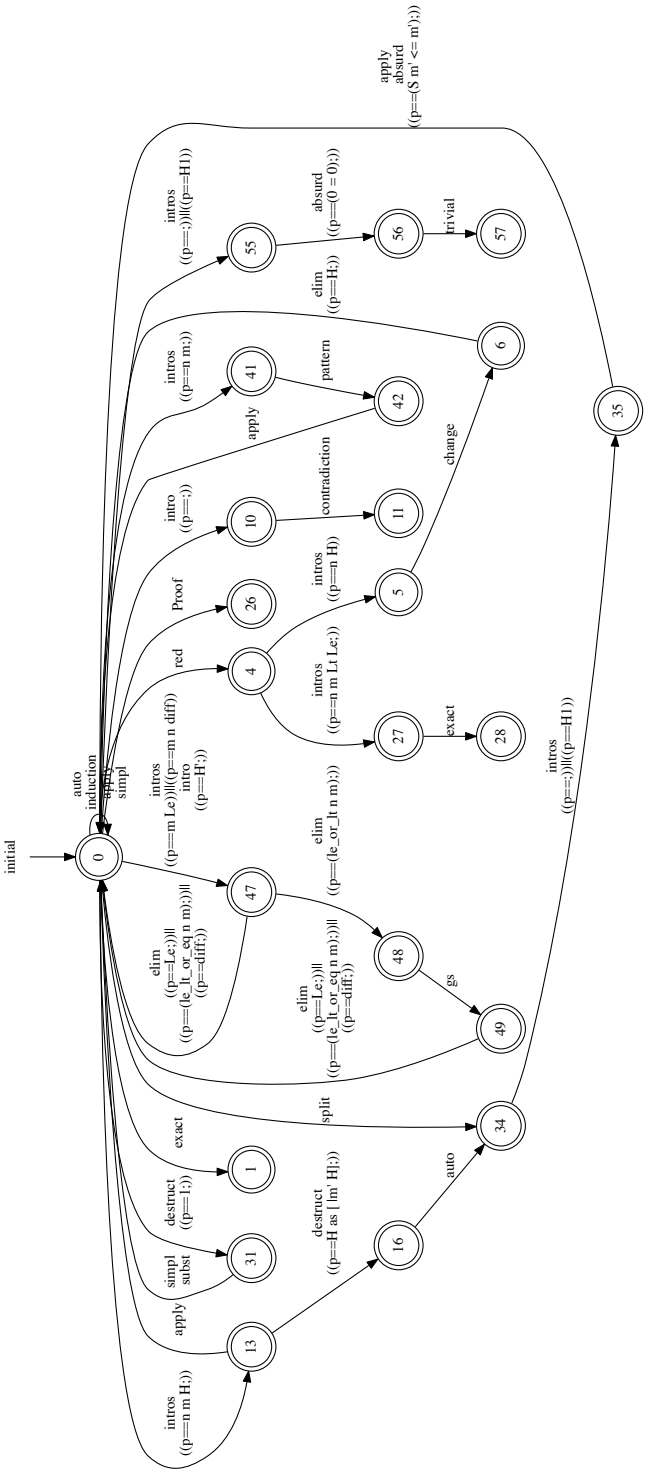


FIGURE 3.6: EFSM inferred from `Le.v` and `Lt.v`

⟨*sentence*⟩ ::= ⟨*assertion*⟩ ⟨*proof*⟩

⟨*assertion*⟩ ::= ⟨*assertion_keyword*⟩ ⟨*ident*⟩ : ⟨*term*⟩.

⟨*assertion_keyword*⟩ ::= `Theorem` | `Lemma`

⟨*proof*⟩ ::= `Proof. ... Qed.`
| `Proof. ... Defined.`
| `Proof. ... Admitted.`
| `Proof. ... Save.`

FIGURE 3.7: Grammar for Coq proof structure

LISTING 3.8: Proof Script to demonstrate parsing in SEPIA

```
Lemma map_app : forall l l', map (l++l')
= (map l)++(map l').
Proof.
induction l; simpl; auto.
intros; rewrite IHl; auto.
Qed.
```

The next part identifies the end of proof. Looking at the grammar, the proof is terminated by either `Qed`, `Defined`, `Save` or `Admitted`. In SEPIA, the implementation works by identifying lines of the theory that are between the `Lemma` or `Theorem` keywords, and the terminating commands. For each proof, this results in providing a string that contains the proof steps used. To demonstrate what SEPIA has identified at this stage, consider the theorem in Listing 3.8.

Everything between the theorem statement and the terminating command (in this case `Qed.`) is stored in a string. For this example "`induction l; simpl; auto. intros; rewrite IHl; auto.`" is stored. As described earlier, the semi-colon operator can be used to form a general sequence of tactics. Any general sequences are split into their individual components. This results in a trace containing 6 elements being created for the example. Importantly, the usage of the semi-colon is recorded within the parameters. The trace generated for the example theorem is shown in Listing 3.9.

3.6 Related approaches

As discussed in Chapter 2, there are existing techniques such as Duncan (Duncan, 2008) and Jamnik *et al.* (Jamnik et al., 2003) who learn from tactic sequences. SEPIA also

LISTING 3.9: Example proof trace

```
trace
induction    l;
simpl    ;
auto
intros    ;
rewrite    IHl;
auto
```

falls into this category by inferring finite state machine models from Coq tactic sequences. There are however some important differences between SEPIA and these alternatives. The learning process in SEPIA differs from that used in existing approaches.

In both approaches there is a process of abstracting away information from the proof sequences. In Jamnik *et al.* the tactic arguments are taken away during the learning process, and re-instantiated during the application of the generated proof methods. In Duncan's work, various different abstractions are studied, many of them losing some aspect of the proof steps such as the direction in which to apply the tactic or the tactic arguments themselves. SEPIA doesn't require any additional abstractions to be made, both the tactic and the arguments are included within the actual learning process.

In Jamnik *et al.* there is the notion of a well-chosen example. This means that by learning from manually chosen groups of proofs, then similar ones can be discharged using the learned tactics sequences. This works well, but only for families of similar proofs that have been manually identified. SEPIA can be used in a more general purpose manner - any collection of proofs can be learned from, and applied to theorems from other domains.

The approach presented by Duncan has some probabilistic elements to it. Tactics are formed from commonly occurring tactic sequences that appear within a corpus of proofs. Although in simple cases this can be useful, the fact that a particular tactic sequence occurs often doesn't mean it can be applied to a lot of other theorems. SEPIA doesn't discriminate based on probabilities and it learns from all tactic sequences provided.

Finally, there are differences in the way that SEPIA learns from the existing proofs. SEPIA treats proofs as static – there is no interaction needed with the theorem prover. Everything is treated as a simple sequence of tactics, and the theory is processed completely textually. In reality, a proof has a branching structure – tactics are applied and create multiple subgoals. Both Jamnik *et al.* and Duncan have to replay the proofs through the theorem prover firstly to acquire the branching structure. This has implications on the speed of the technique. Treating proof scripts textually allows SEPIA to

proceed to the learning process immediately with only simple pre-processing required.

3.7 Conclusions

This chapter has shown how state-based models can be derived from a corpus of interactive proofs. These state machines have proven to be useful as they can reduce large, complex proof files into a more manageable, concise representation. The descriptive models not only show a user the possible sequencing of proof methods (which is valuable enough information on its own), but also help to suggest the parameters that may be useful in completing a proof.

By making two simplifying assumptions about Coq proofs, it becomes possible to quickly infer descriptive models from a corpus of proofs. The first assumption is that a proof is a sequence of tactic applications. In reality, the tactics construct a proof tree – however to generate the proof traces this constraint is ignored. The EFSM inference technique only accepts sequential data, so if the branching structure was considered the proofs would have to be linearized in a manner similar to Duncan's approach (Duncan, 2008). This is a possible item of future work to see if linearizing the proofs leads to better models being inferred. The second simplification is the Coq proof scripts are treated completely textually, leading to information about the tactic arguments being lost. They are simply replayed statically – this potential limitation will be addressed in Chapter 5.

Clearly, this approach is only viable on small collections of examples. In practice, a proof developer might not necessarily want to manually execute a model. One could argue that this approach is no more desirable than manually searching the proof library. However, the example above has shown that SEPIA can discover new links that might not have been spotted from simply analysing the library. The fact that proofs can be found by using SEPIA in this way suggests that if the process could be automated then it could be a useful addition to the proof process.

A small case study demonstrated that the models can be used to derive new proofs. The case study also showed that, in comparison to existing proofs, the EFSM based ones can be shorter than hand-crafted ones. This demonstrates the potential of using state machine inference for theorem proving. The natural progression of this work is to automate this process. This means that no human interaction is currently needed, so there is no need to manually interpret and use the models. SEPIA must be able to interact with the theorem proving tool, and should also be part of the user interface. The next chapter shows how this can be achieved.

Chapter 4

Automating Proofs with Inferred Models

4.1 Introduction

The previous chapter demonstrated that EFSMs can be used manually to generate proofs in Coq. Although useful for small examples, the models inevitably become larger as the corpus size increases. Therefore, it is unrealistic to assume that a user would want to manually use a large inferred model to identify a proof. For the technique to be more appealing to a wider audience, the approach must be as automated as possible.

This chapter describes a technique to automate the process of searching the inferred models. To achieve this, a proof search algorithm is designed that searches the inferred state machine automatically. The algorithm interacts with Coq by executing the tactics encoded within the model. If a proof is identified, the tactics are provided to the user who can simply add them to their proof development. The algorithm forms a key part of the SEPIA plugin – an extension to ProofGeneral that allows one-click invocation of the proof search.

The chapter begins by describing the proof search algorithm in detail. Then, the integration of SEPIA into the ProofGeneral environment is demonstrated. Finally, a walkthrough of the tool shows SEPIA identifying proofs automatically. The work completed within this Chapter is described in a paper presented at CADE-25 (Gransden, Walkinshaw, and Raman, 2015).

4.2 SEPIA Proof Search Algorithm

This section describes the overall process of automating the search for proofs using inferred models. Before describing the proof search algorithm (Algorithm 3), some necessary auxiliary functions are introduced. Following this, the algorithm is described

```
   Input: Theorem statement T, State Machine M
   Output: Coq tactic sequence to prove T
   Data:
1  toVisit ← ∅
2  root ← createNewNodeInformation(∅, getRoot(M))
3  toVisit ← toVisit.add(root)
4  while toVisit is not empty do
5      current ← toVisit.getNext()
6      incomingTactics ← current.getIncomingPath()
7      stateNum ← current.getNode()
8      for (from, label, params, to) ∈ getOutgoingTransitions(M, stateNum) do
9          tacticSequence ← incomingTactics + (label, params)
10         (proven, madeProgress) ← executeCoqCommand(tacticSequence)
11         if proven then
12             return tacticSequence
13         else
14             if madeProgress then
15                 newPath ← createNewNodeInformation(tacticSequence, to)
16                 toVisit.push(newPath)
17             end
18         end
19         resetToInitialState()
20     end
21 end
22 return ∅
```

Algorithm 3: SEPIA Proof Search Algorithm

in depth. As input, it takes a Coq theorem statement that requires proving and an inferred state machine (inferred using the techniques from Chapter 3). As output, the algorithm may produce a Coq command that can be used to prove the theorem.

4.2.1 Auxiliary functions and data structures

The SEPIA algorithm contains some auxiliary methods and data structures that require explanation.

root and newPath variables The algorithm must know which state to visit next in the machine, along with the sequence of tactics that led to that state being encountered. In SEPIA, the `NodeInformation` structure stores these two pieces of information. Algorithm 3 uses variables called *root* and *newPath*, which are instances of NodeInformation. A source code listing for `NodeInformation` is shown in Appendix A.

createNewNodeInformation function During the proof search, some new instances of `NodeInformation` need to be created. This function takes two arguments – a state within the model and a list of incoming tactics that led to the state being encountered. Calling this function results in a new `NodeInformation` being created that stores the tactics and the state.

getOutgoingTransitions function When the proof search visits a state within the machine, the outgoing transitions are explored and the tactics applied to the proof state. This function takes the state machine and the current state. It returns the set of outgoing edges from the state.

executeCoqCommand function The proof search algorithm constructs Coq commands and applies them to an instance of Coq. In return, the method reports back one of three different possibilities. Either the command proved the theorem, or it may have made some progress. Finally, the applied tactics may have caused an error.

resetToInitialState function This function sets the proof state back to the original by issuing the `Restart` command to Coq. This method is important because the proof search algorithm stores the next state and path of tactics that led to it (using the `NodeInformation` class).

4.2.2 Algorithm description

Lines 1-3 The SEPIA algorithm first sets up the environment and any variables. An instance of Coq is created that has states the theorem to be proven. A data structure called `toVisit` stores instances of the `NodeInformation` class defined earlier. Initially, `toVisit` is empty. In practice, precisely what this data structure is depends on the type of search being performed.

If depth-first search is being used, `toVisit` is a stack and conversely using breadth first search it is a queue. SEPIA implements both of these search types – defaulting to breadth-first search. Using the `createNewNodeInformation` method, a `NodeInformation` object is created and added to *toVisit*. This contains the root state of the machine, and an empty incoming path (as there were no tactics applied to arrive at the initial state).

Lines 4-7 The iterative proof search begins by checking if `toVisit` is empty. If it is, then there are no more viable paths to explore within the model and an empty list of tactics is returned. Otherwise, the main loop is entered as further exploration of the model is possible. A variable `current` stores the next `NodeInformation` instance retrieved

from `toVisit`. Other variables, `stateNum` and `incomingTactics` store the incoming tactic path and state number from the *current*.

Lines 8-10 From the state currently being examined (*stateNum*), any outgoing transitions are obtained from the state machine. On each transition, there is a tactic and argument, along with a source ($from$) and destination (to). A Coq command is constructed by pairing a tactic and an argument, and appending this to the incoming tactic sequence to the current state. The resulting command is sent to Coq and the response is then checked.

Lines 11-18 If the theorem was proven, then the successful command is returned to the user. If the tactic was applied and made progress (see below), then a new `NodeInformation` is constructed `newPath` that contains the applied tactic sequence and the destination state of the transition. Then *newPath* is added to the *toVisit* data structure for further investigation later on during the proof search.

Line 19 When the tactic sequence has been evaluated the proof state is reset. This is done by using the `ResetToInitialState` function described earlier.

Checking progress of applied tactics

When SEPIA sends a command to Coq, it also appends the Coq `progress` tactical (Bertot and Castéran, 2004) to the front of it. When investigating a particular path through a model, the aim is to only explore subsequent states whilst progress is being made. To demonstrate why this is important, consider the following example of a potential looping state. It is entirely possible that SEPIA may propose a model that contains a state that has looping transitions (see Figure 4.1).

FIGURE 4.1: Looping transition

It happens that tactics such as `simpl` tactic (among others) don't report an error when no changes are made to the proof state. In this case, the proof search will simply continue searching around the loop (as no error message was given by Coq). To demonstrate this, the example in Listing 4.2 shows the difference between using `progress` and not.

LISTING 4.2: Progress in Coq

```
1 subgoal

============================
forall a : nat, a + 0 = a

demo < simpl.
1 subgoal

============================
forall a : nat, a + 0 = a

demo < simpl.
1 subgoal

============================
forall a : nat, a + 0 = a

continues....
```

By repeatedly applying `simpl` (mimicking the loop displayed in Figure 4.1), the search would simply keep going. This is because the algorithm believes that the tactic was applied successfully. In reality, there are no changes being made to the underlying proof state so there is little point in applying the tactic any more. Now consider Listing 4.3 that shows the same example but with the `progress` tactical being used.

LISTING 4.3: Using the Progress tactical

```
1 subgoal

============================
forall a : nat, a + 0 = a

demo < progress simpl.
Error: Failed to progress.
```

This time, the proof search halts as Coq reports that no progress was made. SEPIA identifies that an error occurred and will stop searching down that particular path. This improves the efficiency of the proof search, as only paths that are genuinely making a difference to the proof state are processed further.

Extending with heuristics

As well as using standard algorithms to traverse the model, the SEPIA approach supports the definition of heuristics that could be used during proof search. This could be a useful addition as the standard search algorithm (Algorithm 3) doesn't include any information that can be obtained from the proof state.

Easily obtained information from the proof context could be, for example the number of subgoals or hypotheses. The length of the incoming path of tactics applied so far could also be considered. By influencing the proof search with this information, the proof search algorithm could select paths based on these properties and explore those first. This idea of proof search heuristics are described as future work in Chapter 7

4.3 The SEPIA ProofGeneral plugin

Previously, models were inferred from Coq proofs and presented to the user. These were then used manually to derive proofs – this involves the user interpreting a model and issuing the suggested steps to Coq. However, by using Algorithm 3, this process can be automated. The remaining task is to make this approach usable during interactive proof development.

To allow SEPIA to be used during proof development, there are some technical obstacles that must be addressed. Firstly, a user interface must be designed to allow the user to invoke SEPIA. Secondly, SEPIA needs to be able to interact with Coq. This involves being able to send commands to Coq, and also receive the response.

The rest of this section is structured as follows: firstly, the overall design and benefits of implementing SEPIA as a Coq plugin are summarised. Then, ProofGeneral is introduced as it forms the basis for the SEPIA plugin. Finally, the process of allowing SEPIA to communicate with Coq is described.

4.3.1 System design and benefits

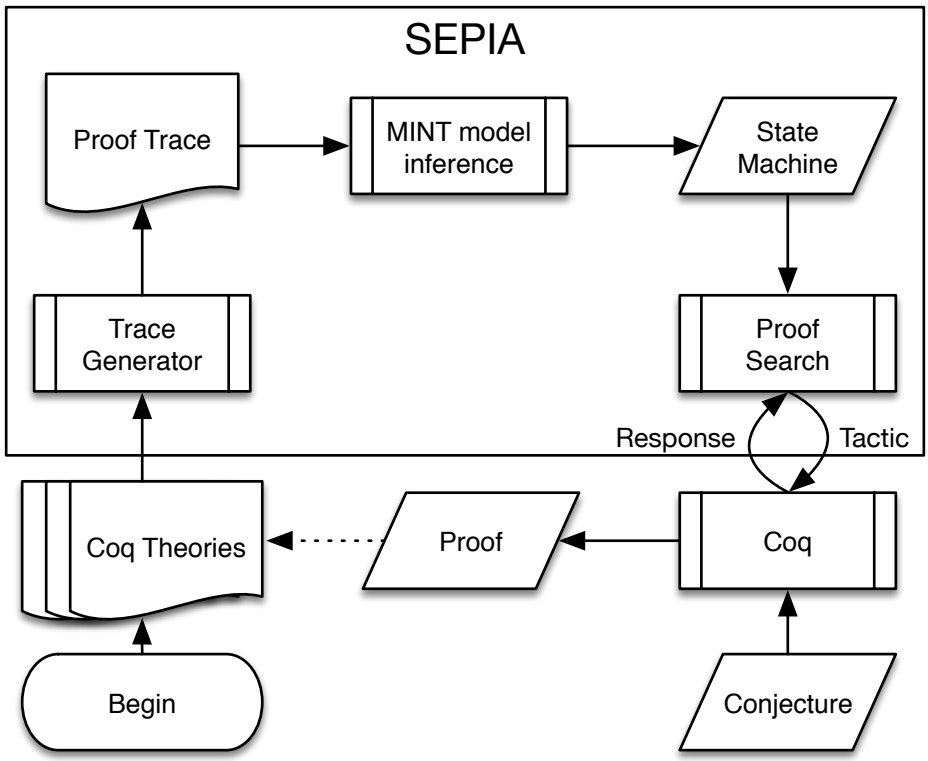


FIGURE 4.4: SEPIA Plugin Overview

Figure 4.4 provides a high level overview of how the components of SEPIA combine to form a usable plugin. It obscures any information about the user interface for now, rather displaying the overall behaviour from starting the process to finding a proof. SEPIA begins with the user selecting the theories that they wish to use for model inference. Then, using the methodology from Chapter 3 a set of proof traces are inferred from the selected theories. Finally, the proof search algorithm (Algorithm 3) is used after a model has been inferred. There are numerous benefits that could make such a plugin attractive for a proof developer:

Adaptivity SEPIA is iterative, meaning that as more proofs are discovered they can be incorporated into future cycles of the process. This could lead to more accurate models being inferred – forming a virtuous loop. This ability to adapt is a major benefit over existing automated tactics that typically try a fixed set of basic tactics and facts.

Automation Aside from providing the initial set of theories from which to infer a model, the user is not prompted for any other input. In addition, the process runs in the

background and alerts the user if a proof is discovered. The user then adds the proof steps into their own proof development. As will be demonstrated in Chapter 6, SEPIA can discover proofs in a matter of seconds.

Shorter proofs The automatic nature of the proof search means that there are many choices of algorithms available. When searching the model, there is a possibility of finding shorter proofs contained within the model (if they exist). When compared with human derivations of the same theorems, the proofs that SEPIA finds may be shorter. The running example from Chapter 3 has already demonstrated this.

New proof sequences The state-merging process (see Chapter 2) can result in models that suggest sequences of tactics that weren't in the initial set of proofs. These aren't necessarily intuitive and may not be spotted from manual scrutiny of the library. Nevertheless, they contain valid sequences of Coq commands that can be used to prove Coq theorems.

4.3.2 ProofGeneral interface for Coq

Coq is a process that uses a traditional read-eval-print loop (REPL) to take commands from the user. It evaluates the command and reports back whether it was successful or not. The downside of such a primitive approach is that a proof development typically spans many files, and there usually needs to be many theories open at once. Therefore, it is desirable to add a user interface layer to help users maintain this information during proof development.

There are many different user interfaces available – these include both web based and ones that can be run locally. For SEPIA to be useful to proof developers, it must be able to be invoked during proof attempts. The most widely used interface is ProofGeneral (Aspinall, 2000). This is a generic emacs interface that can work with multiple popular proof assistants such as Coq and Isabelle. Clearly it is desirable that SEPIA integrates within an existing environment instead of creating one especially for the purposes of this work.

ProofGeneral allows users to create and embed their own plugins within an installation. SEPIA takes advantage of this by implementing a simple plugin that allows the model inference tool to be invoked automatically from within a proof development. ProofGeneral provides a number of convenient mechanisms from a SEPIA perspective, meaning that a new environment doesn't need to be written from scratch.

Firstly, SEPIA can work with multiple Coq theories. These can be provided to SEPIA and used to infer proof traces from (see Figure 4.4). ProofGeneral allows multiple Coq files to be opened and manipulated within the same instance. Therefore, it is trivial to use emacs functions to retrieve the opened theories. ProofGeneral is also able to invoke external processes – the SEPIA executable is in Java format, so it needs to be run from the command line.

SEPIA interacts with an instance of Coq during the proof search. When the user has stated a theorem, an instance of Coq is created that exactly mirrors what the user can see in ProofGeneral. SEPIA interacts with the newly generated Coq instance. This means that if SEPIA finds a proof, the sequence of tactics can be copied into ProofGeneral.

A final benefit of using ProofGeneral is that it is extensible and used with numerous theorem provers. In the future SEPIA may be extended to other theorem proving systems. Having this basic infrastructure in place means that extending should be a trivial process.

4.3.3 Communication between SEPIA and Coq

SEPIA handles communication to and from an instance of Coq. This means that tactics need to be sent to the theorem prover and the response must be provided back to SEPIA. The core implementation of SEPIA is written in Java, whilst Coq is a command line executable (originally programmed in OCaml). Therefore, some careful consideration is needed to get the two tools interacting.

The Coq executable can be launched from Java using the standard Java API methods. This provides a convenient mechanism for invoking and interacting with programs such as Coq. In the ProofGeneral plugin, when the user calls SEPIA an instance of Coq is loaded that mirrors the state that ProofGeneral is currently in.

Once a tactic has been sent to Coq, SEPIA pauses and waits for a response. Coq provides reasonably verbose output, so it is easy to check the result of the tactic application. A simple helper class handles the parsing of the Coq response. There are three possible responses that SEPIA checks for. Firstly, there is `SUCCESS`, which is received if Coq says the theorem is proven. Secondly, there is `OK` which means that the tactic was successfully applied but didn't prove the theorem. Lastly, there is `ERROR`, which signifies that the tactic wasn't applicable to the proof state or some error was output by Coq.

SEPIA reads the output from Coq and scans for keywords in order to decide what the status should be. In the case of `SUCCESS`, this is chosen when the response from Coq contains `Proof completed` or `No more subgoals`. The two success messages are from different versions of Coq so SEPIA checks for both. An `ERROR` status is given

when the response contains a line beginning with the Coq keyword `Error`. Finally, the `OK` status is given in all other circumstances when a tactic was applied successfully to the proof state without error. This mechanism is used in Algorithm 3 to monitor whether tactics were successfully applied.

4.4 Using the SEPIA plugin

To conclude this section, the SEPIA plugin is demonstrated. Firstly, the user can begin their proof development in the usual way. As an example, consider trying to prove the following theorem: `forall n m p:nat, p + n <= p + m -> n <= m.` The theorem has been stated, and the existing Coq automation has been invoked. These tactics were unable to identify a proof for the example. Figure 4.5 demonstrates what the user can see after applying the automated tactics.

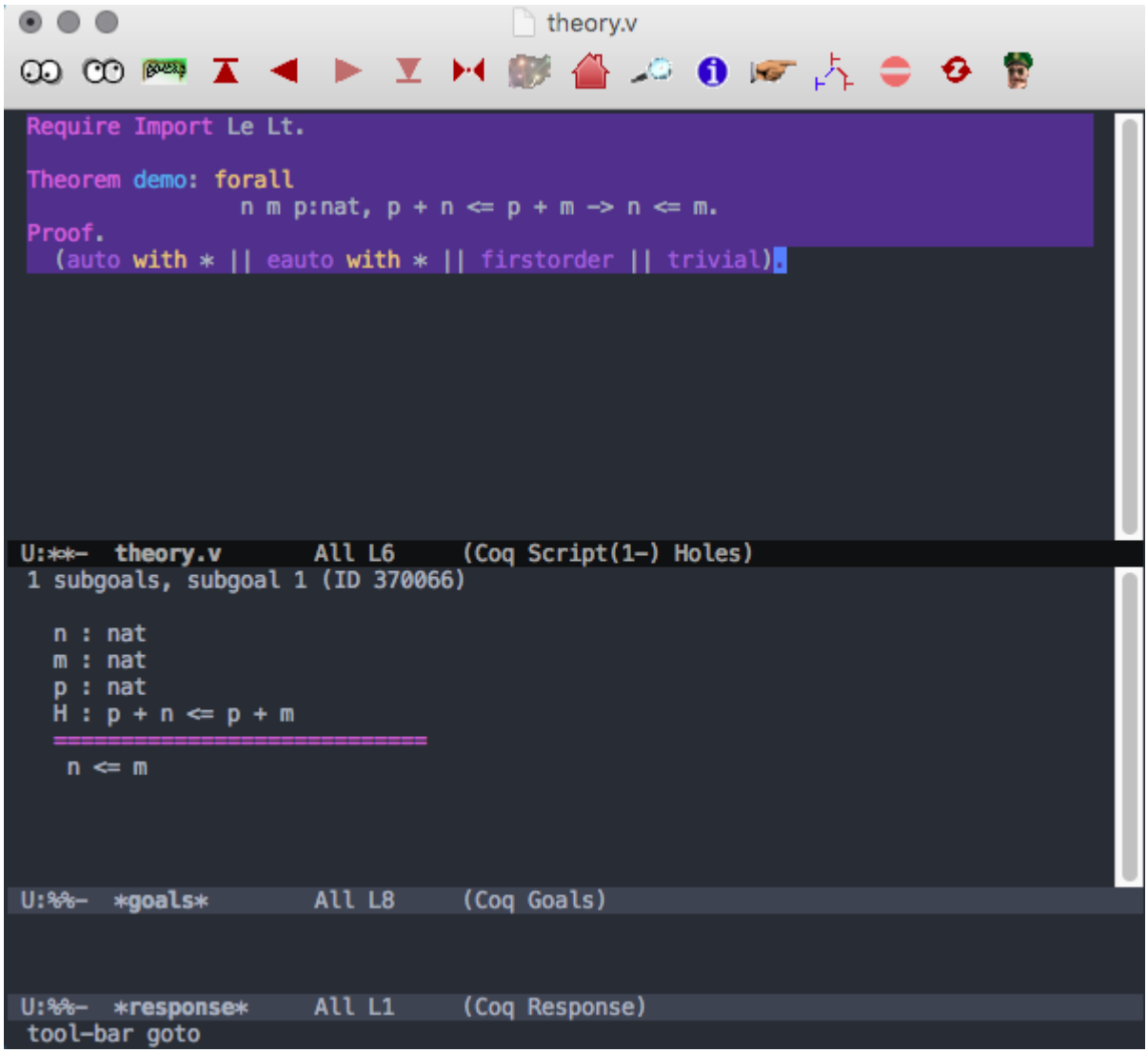


FIGURE 4.5: Stage 1: State the theorem

There are two theories from the Coq standard library containing proofs of similar properties. In order to use these files with SEPIA, they must be opened in ProofGeneral. The user opens up these theories and then returns to their current theory. ProofGeneral is able to list any opened files with a simple call to the `buffer-list` method. Any buffers that don't end with the Coq theory file extension (`.v`) are ignored. Figure 4.6 shows the list of buffers that are open. SEPIA uses any Coq theories that are currently opened.

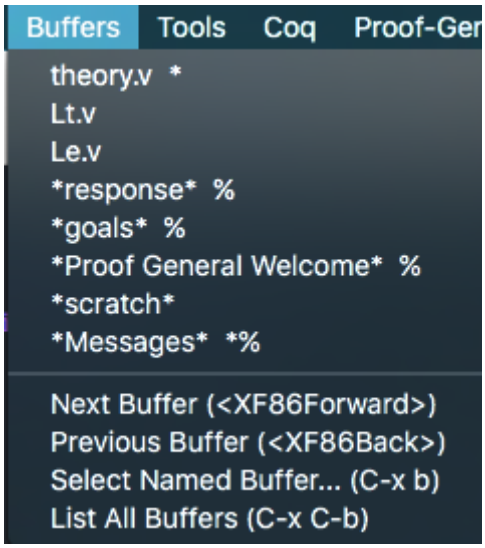


FIGURE 4.6: Stage 2: Open any additional theories

A proof attempt can now begin by invoking SEPIA. The plugin adds an additional menu item to ProofGeneral that allows SEPIA to be configured. The new menu item contains the following options allowing various aspects of the proof search to be changed by the user:

- **Prove current lemma** - this option is selected to begin an automated proof attempt using SEPIA
- **Search type** - a selection of searches can be chosen including breadth-first and depth-first.
- **MINT Options** - For advanced users, some parameters for the MINT tool (Walkinshaw, Taylor, and Derrick, 2015) can be configured. These include selecting the state-merging strategy used, and the score (k) used to determine if two states are equivalent.
- **Search limits** - This allows the proof search to be aborted after a specified number of tactics have been evaluated within the model.
- **Timeout** - This tells SEPIA to stop the search after a specified number of seconds. This provides an alernative to using search limits.

If the user doesn't select any additional configuration options, they are left as their default values. These tell SEPIA to use a breadth-first search for 30 seconds. For this example, everything is left to the default values. MINT is left untouched and uses its standard configuration of state merging and k-value.

After setting any configuration options, the user then clicks the Prove Current Lemma button to begin a proof attempt. Figure 4.7 shows that a new buffer appears called *sepia-output*. This displays any information that is output from the proof search. Green text indicates that a proof was discovered, red text denotes that no proof was found.

At the bottom of the SEPIA output window, the user is provided with some lightweight statistics about the proof attempt. It shows the number of tactics applied during the search, and how long the inference and search took. For the example theorem, 3650 tactics were applied and the overall running time (model-inference and search) took 4 seconds to identify a proof.

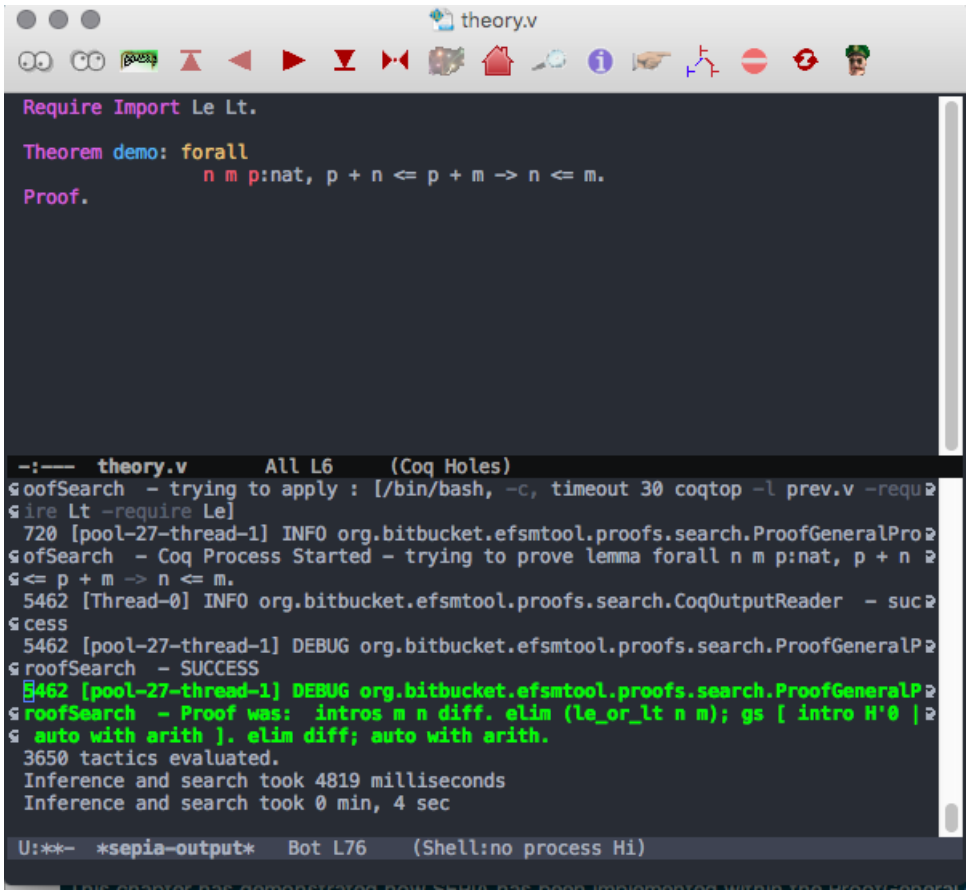


FIGURE 4.7: Stage 3: Invoke SEPIA

For the example theorem, some green text is displayed. This means that SEPIA has proposed a sequence of tactics to prove the theorem. All the user has to do is copy the tactic sequence into their proof development and run Coq to verify the steps. As shown in Figure 4.8, SEPIA has successfully proven the theorem. Interestingly, the proof that SEPIA found was a new combination of tactics not seen in `Le.v` or `Lt.v`.

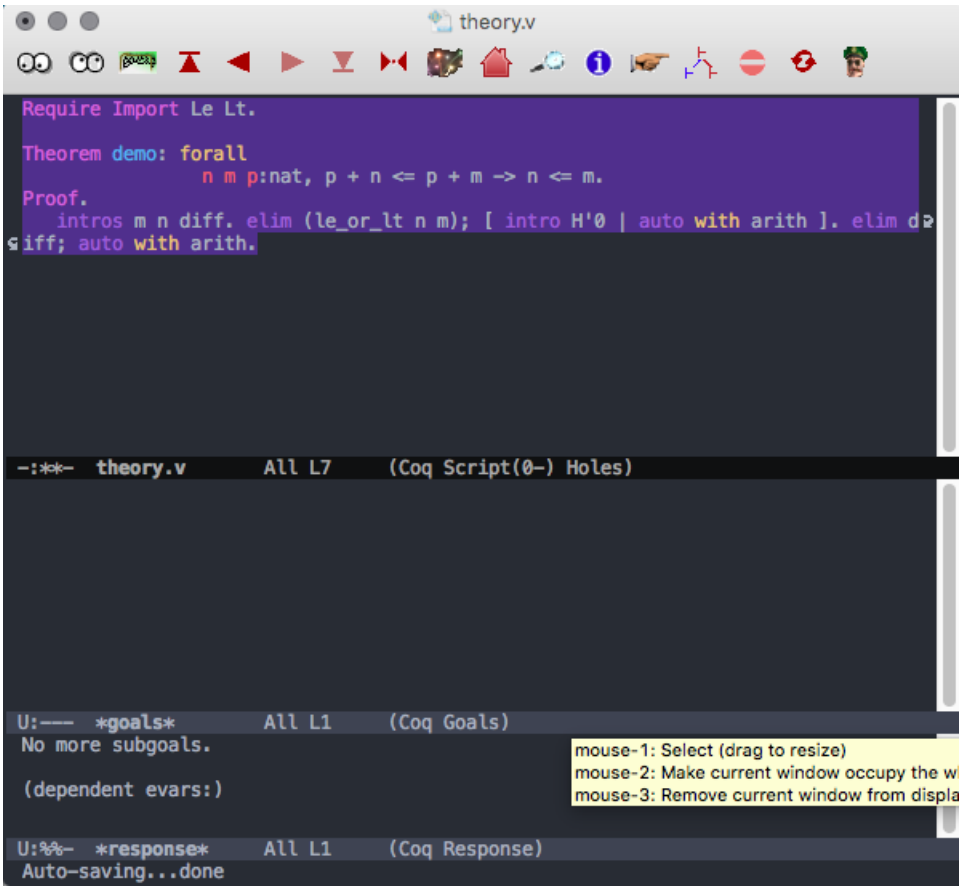


FIGURE 4.8: Stage 4: Paste proof into proof script

4.5 Conclusions

This chapter has demonstrated an automatic proof search algorithm that can execute inferred EFSMs within Coq. A plugin that uses the proof search algorithm has been implemented within the ProofGeneral environment. The extension allows users to select Coq theories, infer models from them, and use the inferred models as a basis for automated proof generation. Aside from selecting the theories to use, the approach is completely automatic. The idea will be fully evaluated later on in Chapter 6.

A walk-through of the tool demonstrated that SEPIA can prove theorems that weren't solved by previously available automated tactics in Coq. This is a major improvement over existing proof automation that is typically limited in the steps that are attempted. Instead of a user studying the model and manually applying the tactics (as shown in Chapter 3), the algorithm achieves this automatically. Ultimately, this means that a model can be searched in a much quicker time than a manual derivation.

The proofs that SEPIA identifies are interesting for a number of reasons. Firstly, as shown in Chapter 3 the proofs can be shorter than a typical human derivation. One

interesting aspect of the discovered proofs is that the model inference techniques can identify new links between different parts of Coq proofs. This was demonstrated by the example from the previous section. Because of the underlying model inference algorithms, SEPIA may suggest entirely new sequences of tactics that weren't present in the original corpus.

Although SEPIA has been demonstrated to be useful at automating proofs, it does have some limitations. The next chapter looks at scenarios where SEPIA is unable to produce proofs, and tries to identify the underlying reasons why a proof wasn't found. To try and address these issues, additional learning steps are introduced to improve SEPIA further.

Chapter 5

Extensions to SEPIA

5.1 Introduction

In the previous chapters, SEPIA has been shown to be a promising approach for automating proofs within Coq. Tactic applications used in existing proofs can be modelled using state-machines, and the resulting models provide a targeted search space that is used during proof attempts. This chapter proposes that additional machine learning can improve the approach further, overcoming some of the limitations that arise from the design choices made in SEPIA.

By taking complete theories of Coq proofs, SEPIA infers a model from *every* proof within the input set. This can lead to large state machine being inferred, that could take a long time to search through. Secondly, SEPIA parses the proof script without any interaction with Coq. Although effective, this choice can limit the applicability of the inferred models. To overcome these limitations, SEPIA is augmented with additional learning steps.

These limitations can be addressed by utilising proof clustering techniques from ML4PG (Komendantskaya, Heras, and Grov, 2013). The addition of the proof clustering forms a suite of machine learning tools[1] called Coq-PR^3. The name derives from three operations that can be performed on Coq proofs – revisiting, reusing and recycling proofs. These will be elaborated on later in Section 5.4.

The chapter begins by describing two scenarios that show potential weaknesses in the SEPIA approach. Then, the ML4PG technique is recapped – focussing on potential areas where it could improve SEPIA. The main algorithm of Coq-PR^3 is described before looking at the corresponding plugin for ProofGeneral. Finally, the examples are revisited to show how Coq-PR^3 can prove additional theorems to SEPIA.

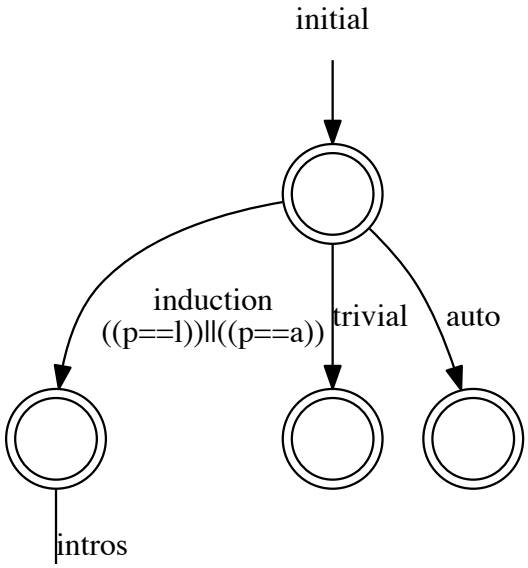
[1]The Coq-PR^3 ProofGeneral extension is available for download from `https://bitbucket.org/tomgransden/coq-pr-3`

5.1.1 Motivating Examples

There are two choices made for the original design of SEPIA that can potentially cause limitations to the technique. These are demonstrated with the help of two examples. The underlying reasons why these situations arise are discussed, along with how additional machine learning can help to address them.

Example 1 - Trace encodings Consider trying to prove the following theorem `forall n m: nat, n + S m = S n + m`. Assume that there is a set of proofs and a model has been inferred from them (the initial fragment is shown below in Figure 5.1). Using the existing SEPIA proof search algorithm (see Algorithm 3), the theorem isn't proven. After investigating the proof search in more detail, it becomes clear why the attempt was unsuccessful.

FIGURE 5.1: Fragment of inferred model from failed proof attempt



The proof search begins in the initial state of the model, and with the theorem stated in Coq. Taking the leftmost transition first, there are two Coq commands that are formulated from this transition – `induction l` and `induction a`. Neither of these can be applied to the proof state, so this path isn't explored further.

Moving to middle transition, there is the `trivial` tactic, that takes no additional arguments. When this tactic is applied an error is returned (recall that SEPIA uses the `progress` tactical) suggesting that no progress was made. Finally, exploring the rightmost transition attempts to apply `auto`. The result is the same as before – no progress was made. There are no more states in the model available to search, so the proof search fails.

It is important to consider why such a proof attempt failed. The automatic tactics failing is not something that can be prevented – either they will make progress or not. The interesting part of this example is the transition labelled `induction`. The application failed because there were two very specific suggestions for arguments – these variables appeared in proofs that the EFSM was inferred from. The tactic application failed because the example theorem contains only `n` and `m`. The transition within the machine states that induction should be performed on `a` or `l` – neither of these are present within the current theorem.

This scenario arises a lot during SEPIA proof attempts, and is caused by the chosen way of encoding proof traces. By enhancing the information contained within the traces, this situation may be overcome. The enhancements to proof trace generation will be described later on in this chapter.

Example 2 - Size of the inferred models SEPIA works by taking one or more Coq theories, and inferring a model from *every* theorem within the input set. There is no relevance filtering (Meng and Paulson, 2009; Kaliszyk and Urban, 2015) that takes place. This naturally means that the inferred models can grow to be extremely large (depending on the proofs used as input). To demonstrate this, consider the following example that uses a theory from the Ssreflect proof development (Gonthier, 2005).

A user has invoked SEPIA to try and prove the following theorem about sequences: `take_size s : take (size s) s = s.` The `seq` theory in Ssreflect contains 393 other proofs about sequences, so SEPIA infers a model from all of these. The resulting model contains 88 states and over 250 transitions. SEPIA manages to prove the theorem, however there may be improvements that can be made to reduce the size of the inferred model.

The reality of inferring from large collections of proofs is that only a handful of the tactic applications may actually useful in any given proof attempt. A large number of paths through an inferred model simply aren't applicable to a particular theorem. A potential solution to circumvent this situation is to identify which theorems may be useful to infer a model from. By doing this, the models become smaller and more easily searchable within a given time constraint.

By reducing the size of the model (but hopefully preserving the ability to prove the theorem), there can be improvements to the proof search. There will typically be far fewer paths to explore, meaning there are fewer interactions with Coq needed. Also, using a smaller model there could be improvements to the time needed to search the model.

5.2 ML4PG revisited

ML4PG (Komendantskaya, Heras, and Grov, 2013) is a tool for Coq that uses clustering techniques to identify similarities between Coq theorems and definitions. It has successfully been used in a variety of situations (Heras and Komendantskaya, 2014; Heras and Komendantskaya, 2013) to aid proof development. Ultimately, it is used to present proof patterns to the user who can then formulate a proof manually. In Chapter 2, the theory behind ML4PG was described. This section attempts to identify potential ways it can help address the limitations from the previous section.

ML4PG can be invoked in two different modes depending on the situation. Firstly, it can take large proof libraries and cluster them to provide groups of similar theorems. Secondly, ML4PG can be invoked during a proof attempt to gather the most similar theorems to the current one being proven. Both of these modes will be exploited by Coq-PR^3.

	SEPIA	ML4PG
Aim	Automatic proof generation	Provide hints to user
Type of learning	Model inference	Coq object clustering
User interface	ProofGeneral	ProofGeneral
Output usage	Manual/Automatic	Manual
Learning process	Static	Dynamic

TABLE 5.1: Comparison of SEPIA and ML4PG

Table 5.1 provides a comparison of SEPIA and ML4PG. The two tools are both ProofGeneral extensions, but have differing aims. SEPIA focusses on automatic proof generation using model inference techniques. ML4PG uses proof clustering methods to produce proof hints for the user. To achieve this, ML4PG dynamically learns from proofs by interacting with Coq to extract features, whereas SEPIA treats the proof scripts completely statically.

Despite having numerous differences, the two tools can potentially complement each other. Consider the two limitations of SEPIA described earlier. By utilising the functionality provided by ML4PG, the new Coq-PR^3 technique can use the best of both approaches to prove theorems in Coq.

The first limitation of SEPIA was that proof-trace generation simply parsed the proof scripts without interacting with Coq. ML4PG takes a different approach, replaying each proof through Coq to generate features. These features are then passed to clustering algorithms in order to identify similarities between Coq objects. There is a

Input: A Coq theorem T, A corpus of Coq proofs $Proofs$
Output: A tactic sequence to prove T

1 $suggestions \leftarrow getSimilarTheorems(T, Proofs, Granularity)$
2 $(clusters, hypotheses) \leftarrow gatherProofLibraryInformation(Proofs, Granularity)$
3 $Tr \leftarrow \emptyset$
4 **for** $p \in suggestions$ **do**
5 $\quad Tr \leftarrow Tr \cup toProofTrace(p, hypotheses, clusters)$
6 **end**
7 $M \leftarrow inferModel(Tr, suggestions)$
8 $solution \leftarrow performProofSearch(M, T, Clusters)$
9 **return** solution

Algorithm 4: Integrating proof clustering and model-inference

possibility that augmenting some of this functionality into the proof trace generation could produce better trace encodings.

The second limitation was the lack of relevance filtering within SEPIA. ML4PG could trivially be employed as a relevance filter for the model-inference part of SEPIA. Instead of inferring a model from every proof provided, ML4PG could be invoked to make suggestions about which theorems *could* be most useful to use during model inference.

The rest of this section looks at how the two tools (SEPIA and ML4PG) can complement each other. The resulting combination – Coq-PR^3 – provides a new proof search approach that combines model inference and proof clustering. Additionally, the user benefits from being to invoke each tool individually.

5.3 The Coq-PR^3 algorithm

Coq-PR^3 is the combination of the state-machine inference techniques from SEPIA, and the proof clustering methods from ML4PG. The main steps of the algorithm that powers Coq-PR^3 is described in Algorithm 4. As input, it takes a set of Coq proofs ($Proofs$) and a theorem to prove (T). As output, a Coq command that proves the theorem will be provided if one was found. The main steps of the algorithm are described before the changes to proof trace generation and proof search are described.

Line 1 - Identify similar theorems Coq-PR^3 will be invoked once the user has stated a conjecture in Coq. The first stage of the process is to use the proof clustering algorithms to suggest similar theorems to the theorem. The number of suggestions provided is controlled by the specified granularity parameter. If this is low, then a larger set of suggestions is provided. Increasing the granularity reduces the number of suggestions to only provide the most closely related ones to the current theorem.

Line 2 - Gather information from the proof corpus The previous step identified the most similar theorems to the current theorem. Coq-PR^3 must also identify the groups of theorems that are most related to each other across the whole corpus of proofs. Therefore, the proof clustering algorithm is invoked using the complete set of proofs ($Proofs$). Again, the granularity parameter controls the overall size and precision of the clusters.

To enable proof clustering, features are extracted dynamically from Coq proofs. This involves replaying every step through Coq to identify the necessary features. Using this feature extraction mechanism provides the ability to inspect the internal hypotheses. The proof clustering algorithm stores how the hypotheses change between tactic applications. Section 5.3.1 shows an example of hypotheses extraction.

Lines 3-6 - generate proof traces The next step is to generate the proof traces. In Coq-PR^3, the traces are enhanced using information from the clusters and hypotheses that were extracted earlier. By extending the proof trace generation process, it is possible to encode extra information within the traces. This allows better identification of precisely what a tactic argument refers to. The updated proof-trace generation process is described fully in Section 5.3.2.

Line 7 - infer the model Once the proof traces have been generated, the EFSM is inferred. The underlying model inference algorithms in MINT are unchanged. However, the main difference in Coq-PR^3 is that the model is not inferred from all of the proofs. Instead, the suggestions are fed into the model-inference, meaning that only these theorems should be used to infer a model from.

Lines 8-9 - Search the model Because the format of the proof traces has been modified, the proof search algorithm must be adapted to accommodate this. The fundamental approach is the same as the algorithm used in SEPIA (see Algorithm 3). However, the clusters generated earlier are also provided to the search process to enhance the number of choices for possible tactic arguments. The changes made to the proof search are described in Section 5.3.3.

5.3.1 Obtaining hypotheses from Coq proofs

As tactics are applied to the proof state, the context is maintained by Coq. This contains hypotheses that can be used to prove the current subgoal. Tactic arguments may refer to facts available within the context. Everything contained within the context has a name

and an associated statement. In Listing 5.2, the context is everything above the line, with the subgoal displayed below the line.

LISTING 5.2: Coq proof state

```
1 subgoal
n : nat
m : nat
============================
n + S m = S n + m
```

The feature extraction technique used in ML4PG is able to produce a hypotheses list for each theorem it processes. This simply outputs the variable names that were in scope at each step of the proof. The actual statement that the variable refers to isn't important for the purposes of Coq-PR^3 and is ignored. Figure 5.3 demonstrates the hypotheses list for an example theorem called `two_power_nat_pos`.

(a) Proof Script

```
Lemma two_power_nat_pos :
       forall n : nat, two_power_nat n > 0.
Proof.
induction n.
rewrite two_power_nat_0.
omega.
rewrite two_power_nat_S.
omega.
Qed.
```

(b) Hypotheses

```
("two_power_nat_pos"
       (nil
       nil
       nil
       ("n" "IHn")
       ("n" "IHn")))
```

FIGURE 5.3: Coq proof and extracted hypothesis

The output should be interpreted as a list that shows how the context changes after tactics are applied to the proof state. Before any tactics have been applied, the context is empty (`nil`). After applying `induction n`, the context is still empty. Similarly, after applying `rewrite two_power_nat_pos` the context remains empty. Applying the Coq automated `omega` tactic solves the base case of this proof and leaves the step case. The context now contains `n` and `IHn`. The application of `rewrite two_power_nat_S` leaves the context unchanged, before `omega` finishes the proof.

5.3.2 Modifying proof trace generation in Coq-PR^3

To generate proof traces in Coq-PR^3, the raw proof script isn't suitable on its own, and some additional input is required. The main difficulty when parsing the proof script is identifying what constitutes a tactic parameter and what is simply Coq syntax. In

SEPIA, everything after the tactic was simply stored as a string. However, Coq proof scripts can be converted into HTML, providing a more structured format. Consider the following proof – shown in both proof script and HTML format (tidied up into a readable format):

(a) Proof Script

```
Lemma two_power_nat_pos :
forall n : nat, two_power_nat n > 0.
Proof.
induction n.
rewrite two_power_nat_0.
omega.
rewrite two_power_nat_S.
omega.
Qed.
```

(b) HTML Format

```
<span class="id" title="keyword">Lemma</span>
<span class="id" title="var">two_power_nat_pos</span> :
<span class="id" title="keyword">forall</span>
<span class="id" title="var">n</span> :
<span class="id" title="var">nat</span>,
<span class="id" title="var">two_power_nat</span>
<span class="id" title="var">n</span> &gt; 0.
<span class="id" title="tactic">induction</span>
<span class="id" title="var">n</span>.
<span class="id" title="tactic">rewrite</span>
<span class="id" title="var">two_power_nat_0</span>.
<span class="id" title="tactic">omega</span>.
<span class="id" title="tactic">rewrite</span>
<span class="id" title="var">two_power_nat_S</span>.
<span class="id" title="tactic">omega</span>.
```

FIGURE 5.4: Proof script and HTML representation

Using the HTML structure, it becomes much simpler to identify what constitutes the arguments and what is the Coq syntax. All potentially useful information is contained within `<span>` tags. These either describe that the enclosed object is a Coq keyword, a Coq tactic or a variable – it is these variables that are of interest to Coq-PR^3. All syntactical information is contained outside of these tags.

In Coq-PR^3, the proof traces should be as descriptive as possible, adding extra meaning to what the tactic arguments actually represent. For instance, is the argument another theorem, a hypothesis or does it occur in the theorem statement. Using this newly obtained information it becomes possible to assign placeholders for tactic arguments based on information obtained previously. The two placeholders that Coq-PR^3 uses are `<local>` and `<ext-fact >`.

Incorporating local variables To demonstrate using the `<local>` placeholder, consider the Coq proof that has been used throughout this section (see Figure 5.3). The first tactic in the example proof is `induction n`. In the original proof trace generation, this was simply recorded as the proof method `induction` and the parameter as "`n`". The resulting transition within the EFSM can then only be re-applied if there happens to be some variable named `n` in the subgoal (in the same manner as the first motivating example).

Imagine that the proof trace generation is looking at the first line of the example proof – `induction n`. Using the hypotheses generated by ML4PG, the context is checked to see the state immediately before `induction n` was applied. In this case, it was empty so the theorem statement is examined for possible variables. There is a matching variable that came from the original statement – `n`.

This can then be encoded in the trace as follows: the proof method is `induction` and the parameters are stored as "`<local>`". This can be understood by the proof search algorithm as "apply induction on some variable in the context/subgoal". The proof search algorithm can use this guidance to make a choice about how to apply the tactic based on the proof attempt at the time.

Incorporating External facts Coq-PR^3 can also provide a placeholder for external facts – allowing the proof search to try similar theorems. In the example proof, consider the second tactic application – `rewrite two_power_nat_0`. In SEPIA, the proof method is stored in the trace as `rewrite` and the arguments are `two_power_nat_0`. This means that the tactic is only applicable if there is a suitable proof state to rewrite using the specified theorem.

As an example, the clusters provided by ML4PG suggest that there are 12 similar facts to `two_power_nat_0`. These could also be applied during a proof attempt. This is encoded in the proof trace as follows: the proof method is `rewrite` and the parameters are `<ext-fact two_power_nat_0`. The `<ext-fact>` placeholder should be understood as "try using the suggested fact, but also try the replacements available".

Comparing proof-trace encodings Using the enhanced proof trace generation for Coq-PR^3, the example theorem gets encoded as shown in Figure 5.5(b). As a comparison, Figure 5.5(a) shows the same theorem encoded using the proof trace format from Chapter 3. The main fundamental difference is that Coq-PR^3 incorporates placeholders within the proof trace. These are used and instantiated later during the proof search.

(a) SEPIA proof trace

```
trace
induction   n
rewrite   two_power_nat_0
omega
rewrite   two_power_nat_S
omega
```

(b) Coq-PR^3 proof trace

```
trace
induction   <local>
rewrite   <ext-fact two_power_nat_0>
omega
rewrite   <ext-fact two_power_nat_S>
omega
```

FIGURE 5.5: Comparison of SEPIA and Coq-PR^3 proof traces

```
   Input: Theorem statement T, State Machine M, Proof Clusters PC
   Output: Coq tactic sequence to prove T
 1 toVisit ← ∅
 2 root ← createNewNodeInformation(∅, getRoot(M))
 3 toVisit ← toVisit.add(root)
 4 while toVisit is not empty do
 5     current ← toVisit.getNext()
 6     incomingTactics ← current.getIncomingPath()
 7     stateNum ← current.getNode()
 8     for (from, label, params, to) ∈ getOutgoingTransitions(M, stateNum) do
 9         for possibleInstantation ∈ enumeratePossibilities(params, PC) do
10             tacticSequence ← incomingTactics + (label, possibleInstantiation)
11             (proven, madeProgress) ← executeCoqCommand(tacticSequence)
12             if proven then
13                 return tacticSequence
14             else
15                 if madeProgress then
16                     newPath ← createNewNodeInformation(tacticSequence, to)
17                     toVisit.push(newPath)
18                 end
19             end
20             resetToInitialState()
21         end
22     end
23 end
24 return ∅
```

Algorithm 5: Coq-PR^3 Proof Search Algorithm

5.3.3 Enhancing proof search in Coq-PR^3

The final modification required for the Coq-PR^3 approach is to the proof search algorithm. Although mostly similar to the one used in SEPIA (Algorithm 3), there are some changes necessary. These are because the proof traces now contain placeholders, and these are instantiated by the proof search algorithm. A new auxiliary function is described, before the changes to the proof search algorithm are described.

Auxiliary functions

Algorithm 5 uses the same auxiliary functions as before. However there is one additional function required. The `enumeratePossibilities` function takes a parameter string, and performs any instantiations of the placeholders contained within it. The function returns a set of strings that constitute all of the possible instantiations of the parameters.

If the parameters don't contain `<local>` or `<ext-fact>`, then the original string is simply returned as it can be applied to Coq without any additional processing. If the parameter contains a `<local>` placeholder, then Coq-PR^3 inspects the current proof context and subgoal and obtains the names of any variables present. Then, for each variable an instantiated Coq command is constructed and added to the set of strings to return.

Similarly, if the parameter contains an `<ext-fact>` placeholder, the clusters are used to lookup alternatives to the original fact. Again, these are all used to instantiate the placeholder and added to the set to return. In the case that the parameters contain more than one placeholders, then the Cartesian product of the possible instantiations is returned.

Algorithm description

Lines 1-7 The Coq-PR^3 algorithm follows the same basic process as the SEPIA implementation. An instance of Coq is created that has stated the conjecture to be proven. A data structure called `toVisit` stores instances of the `NodeInformation` (see Chapter 4). Initially, `toVisit` is initialised to be empty, however a `NodeInformation` object is created and added to *toVisit* that contains the root state of the machine, and an empty incoming path. The proof search enters the main loop and begins exploring from the root of the inferred machine.

Lines 8-10 From the current state, the outgoing transitions are obtained. On each transition, there is a tactic and argument. The modified proof trace format for Coq-PR^3 means that the parameters may contain placeholders that can be instantiated during the proof search. This step of the algorithm takes the parameters from an outgoing transition, and checks for the presence of `<local>` or `<ext-fact>` placeholders. This is done by the *enumeratePossibilities* function described earlier.

For each of the possible instantiations obtained from *enumeratePossibilities*, a Coq command is constructed. This uses the incoming tactic sequence as a prefix, and the tactic with (possibly) instantiated argument. The resulting command is sent to Coq and the response is then checked.

Lines 11-20 If the Coq command proved the theorem, then the tactic sequence is returned. If the command made progress, then the destination state of the applied command and the tactic sequence is added to *toVisit* for further exploration. After the command has been examined, the proof state is reset back to the initial state. In the case that the model cannot be explored further, then an empty sequence is returned.

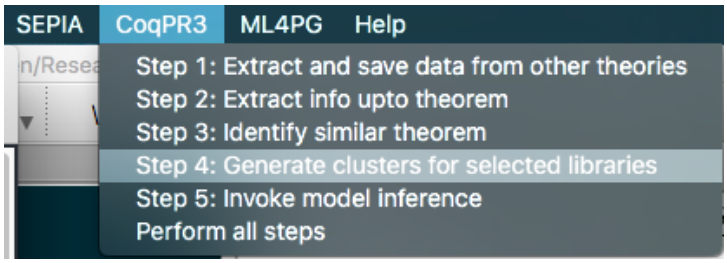


FIGURE 5.6: Coq-PR^3 menu option in ML4PG

5.4 Integrating Coq-PR^3 into ProofGeneral

Coq-PR^3 has been designed to provide proof developers access to a suite of intelligent tools for Coq. There are three different use cases for Coq-PR^3 that a user may wish to invoke. These possible uses of Coq-PR^3 are:

- **Revisiting proofs** - use ML4PG to identify proof-patterns in a library of proofs, and manually formulate a proof using the generated clusters.
- **Re-using proofs** - use SEPIA (described in Chapters 3 and 4) to generate a state-machine that can automatically prove theorems.
- **Recycling proofs** - combine both techniques using Algorithms 4 and 5 to explore theories and automatically prove theorems.

In order to revisit or reuse proofs, the user can navigate to the menu of the respective tool (either ML4PG or SEPIA). In the case that the user wants to try to recycle proof patterns, a new menu item for Coq-PR^3 has been created. The overall process of recycling proofs using Coq-PR^3 is displayed in Figure 5.7.

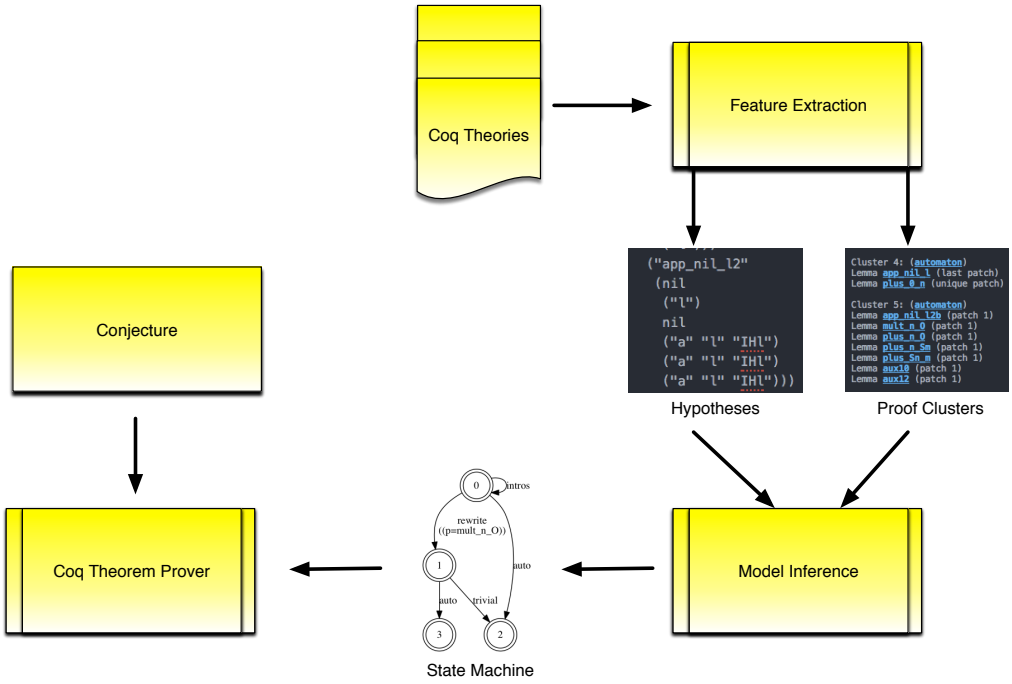


FIGURE 5.7: Recycling proof patterns with Coq-PR^3

The additional Coq-PR^3 menu that allows the user to invoke the proof search algorithm (Algorithm 5). There are numerous steps to perform – these can be done individually or combined into one fully automated step. In order to recycle proofs with Coq-PR^3 the following steps must be performed:

1. **Extract and save data from chosen theories** – There will typically be other theories that the user will have open in ProofGeneral. These need to have their features extracted in order for ML4PG to cluster the proofs. The information for each theory is extracted – both features and hypotheses – and stored for usage later.

2. **Extract info upto theorem** – Using the current theory, there may be theorems that are already defined before the current one. This step extracts the features and hypotheses from these theorems. This data is also stored for usage later on in the process.

3. **Identify similar theorems** – Using all of the proofs that have had their features extracted (using the previous two steps) ML4PG is used to identify the most similar theorems to the current one.

4. **Generate clusters for selected libraries** – ML4PG is used to generate the clusters from all of the proofs selected by the user. These are provided to the proof

trace generation process, and are used as alternative tactic arguments during proof search.

5. **Invoke model inference** – Finally, the model inference from SEPIA is invoked. The suggestions from step 3 are used to tell SEPIA which proofs to infer a model from. The proof traces are generated using the hypotheses (steps 1 and 2) and clusters (step 4) generated in earlier steps (1) of the algorithm. After the model has been inferred, it is searched using the proof search algorithm.

The output uses the same mechanism as SEPIA – an output window opens and provides information about the proof search. If a proof is discovered, the user is provided with the tactics to paste into their proof development. If a proof wasn't discovered, the user can choose to invoke the process again with different settings (e.g. by changing the granularity or the model inference parameters).

Each tool can be configured using their respective menu. SEPIA can be configured using the menu item described in Chapter 4. The existing implementation of ML4PG has a menu item that allows the granularity and clustering algorithm to be configured.

5.5 Examples

To demonstrate the benefits of the Coq-PR^3 approach, the examples described at the start of this chapter are revisited.

5.5.1 More descriptive proof traces

The enhanced models that Coq-PR^3 produces can be searched automatically, instantiating the transitions where appropriate. Transitions within the inferred state machines now contain placeholders telling the proof search what certain tactic arguments are - a local variable, or referring to a fact that can be replaced with information from the proof clusters.

Consider the example proof from earlier in this section where SEPIA attempted to prove `forall m n:nat, n + S m = S n + m`. The proof search failed because there were suggestions in the model to perform induction on variables `l` or `a`. However, in the target theorem there were no such variables - only `n` and `m`. Using the new Coq-PR^3 trace format and proof search algorithm, the same initial fragment of the model is shown in Figure 5.8.

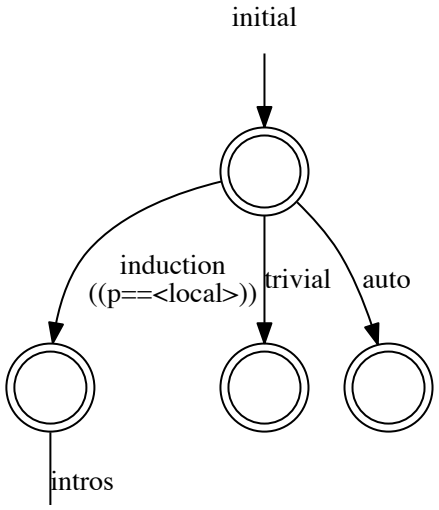


FIGURE 5.8: Fragment of semantic model

Using the search algorithm (Algorithm 5) the semantic model is searched (in this case a breadth-first manner). The leftmost transition is investigated first. This time, the proof search finds a `<local>` placeholder and instead of trying concrete suggestions, it searches the current proof state.

If the search finds any local variables or variables within the subgoal, it attempts to use them. Two such variables are found within the subgoal (the context is empty at this point): `n` and `m`. Two Coq tactics are constructed and applied to the proof state – `induction n` and `induction m`. This time, the proof can progress past the initial state because each of these tactics made progress instead of failing to be applied.

After being able to explore the model further, Coq-PR^3 eventually reports back that a proof was found for the example theorem. The proof discovered was `induction n. intros. trivial. intros. simpl. auto`. Although a simple example, the benefits of Coq-PR^3 immediately become clear. The initial limitation of the proof trace encodings in SEPIA has been overcome. This happens by allowing the proof search to instantiate the tactic arguments, instead of trying hard coded arguments.

5.5.2 Reducing state space

The next example looks at how using the proof pattern recycling methods of Coq-PR^3 can reduce the size of the inferred models. In the second motivating example from the start of this chapter, SEPIA was being used to prove the `take_size` theorem. Using 393 other proofs, a very large model was inferred and searched.

By using the recycling function in Coq-PR^3, the overall size of the model can be reduced dramatically. Asking ML4PG for suggestions similar to `take_size` yields 33 other theorems. Instead of inferring a model from the whole theory of proofs, Coq-PR^3

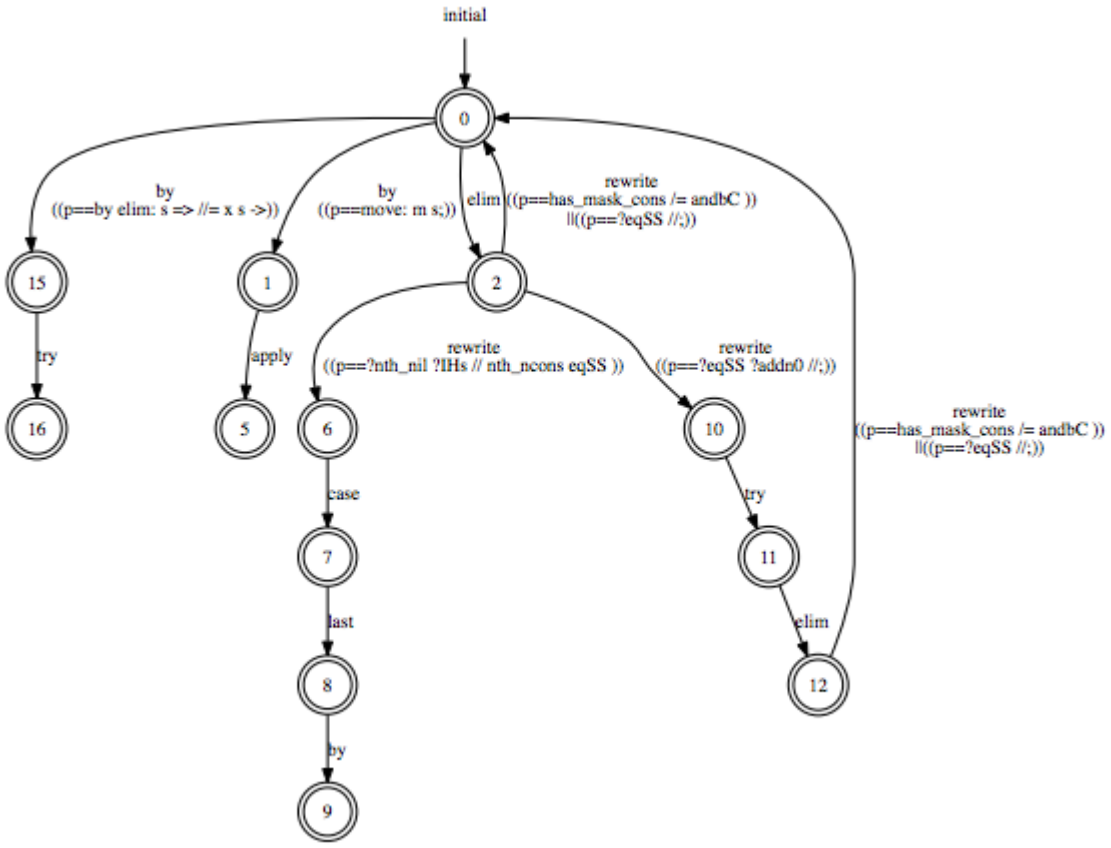


FIGURE 5.9: Reduced state machine inferred from `seq` theory

uses these suggestions to infer the model. Figure 5.9 shows the inferred EFSM. This time, the EFSM has 13 states and much fewer transitions to explore.

Coq-PR^3 identifies the following proof from the EFSM - `by elim: s => //= x s ->`. Although a fairly trivial example, this demonstrates the potential benefits of using relevance filtering to identify the best theorems to infer a model from.

5.6 Conclusions

This chapter has introduced the Coq-PR^3 technique – providing a suite of intelligent tools for proof development. As the combination of two existing tools, the user can benefit from each one individually as well as in conjunction with each other. In order to obtain proof clusters and hints, ML4PG is used. To utilise model inference for automatically proving theorems, SEPIA can be invoked. However the main contribution of this chapter is a novel approach that combines both to automatically prove theorems in Coq. The underlying algorithm that powers Coq-PR^3 has been described, along with modifications needed to improve the SEPIA approach.

The Coq-PR^3 approach has been implemented as a ProofGeneral plugin. A description of the main features of this plugin has been provided. There are three main functions that a user can perform using the plugin. Revisiting and re-using proofs invokes ML4PG and SEPIA as individual tools with no interaction between each other. A third way to use Coq-PR^3 allows the proof patterns from both tools to be recycled for the purposes of automated proof generation in Coq.

The combined approach addresses some limitations of both SEPIA and ML4PG. The design choices of SEPIA mean that the inferred models were potentially too specific. This means that the tactics contained within the model could only be applied in specific situations. By combining with ML4PG, the models become more generally applicable. Additionally, by employing ML4PG as a relevance filter, the overall size of the models can be reduced but theorems can still be proven.

In terms of ML4PG, the original output produced by the tool had to be used manually. When combining with SEPIA, this restriction is lifted as a model can be inferred from the suggestions that ML4PG produces. Overall, both tools have been by forming Coq-PR^3 from the two different approaches. Crucially, the combination has the potential to prove more theorems automatically than before.

Chapter 6

Evaluation

6.1 Introduction

The previous chapters have described two techniques designed for proving theorems automatically in Coq. SEPIA uses state-machines to model existing Coq proofs. The resulting models can be searched automatically by executing the tactics encoded within the model. Coq-PR^3 combines SEPIA with a proof clustering tool (ML4PG) to try and address some limitations of the SEPIA technique. Both techniques have been implemented as plugins for ProofGeneral that are used for automated proof generation in Coq.

Up to this point, the tools have been demonstrated on smaller case studies. In this chapter, the two techniques are rigorously evaluated on a selection of existing Coq proofs. Two large well-known Coq datasets are used as a basis for the evaluation. These provide a diverse selection of proofs to evaluate both SEPIA and Coq-PR^3 on. Various research questions arise that will (when answered) address various aspects of the techniques, and provide an indication about whether the tools could be useful in practice. In addition to the techniques being evaluated based on their ability to prove theorems, the proofs discovered by the tools are also investigated.

The chapter begins by defining the research questions that this evaluation will attempt to answer. The two datasets used throughout this evaluation are also discussed. Following this, for each question an associated experiment and methodology is described. Finally, the results from the experiments are presented and discussed.

6.1.1 Research Questions

The effectiveness of the two techniques presented in this thesis can be measured in terms of the following research questions:

- RQ1 - Is SEPIA able to prove theorems automatically in Coq?

- RQ2 - Does SEPIA complement the existing automated tactics that are available in Coq?
- RQ3 - Are there any interesting properties about the proofs that SEPIA discovers?
- RQ4 - Does the Coq-PR^3 approach enhance SEPIA by proving additional theorems?

6.2 Methodology

There are numerous experiments that have been conducted in this evaluation. To obtained answers to the above research questions, four experiments have been designed. This section describes the methodology used in each experiment, and what the results will potentially indicate about SEPIA and Coq-PR^3. Two datasets are used throughout to maintain consistency between the experiments. These are described in further detail below.

6.2.1 Data Sets

To evaluate SEPIA and Coq-PR^3, two diverse Coq proof developments are used. There were many properties of the datasets that made them appealing to use. Ssreflect and CompCert were selected due to their size, varying complexity, subject matter and input language. Additionally, the two datasets use different proof languages . Some additional details are given below.

- **Ssreflect core library**[1] The Ssreflect library (Gonthier, 2005) contains a selection of theory files covering basic mathematical concepts. There are 1389 theorems contained in 8 theory files. The proofs are written in a variation of the standard Coq dialect called Srefelect, and is a language that allows small-scale reflection. This is a style of proof adopted by Gonthier *et al* in their developments of the Four Color and Odd Order theorem proofs.
- **CompCert**[2] The CompCert development (Leroy, 2009) contains proofs that certify that a C compiler doesn't introduce any errors during compilation. As such, there is a varied selection of theories and proofs available. There are 69 theories containing 3248 theorems in total. This proof development uses the standard Gallina (Bertot and Castéran, 2004) input language for Coq.

[1] http://ssr.msr-inria.inria.fr/doc/ssreflect-1.5/
[2] http://compcert.inria.fr/doc/index.html

6.2.2 Attempting proofs with SEPIA

The aim of the first research question is to investigate whether SEPIA is able to prove theorems in Coq. In order to estimate the usefulness of SEPIA, consider the following situation: given some existing proofs, can these be used by SEPIA to prove new properties that aren't part of the initial collection. This experiment helps to evaluate the SEPIA proof search algorithm (Algorithm 3), and is performed on a per-theory basis (using the theories from both datasets described earlier).

To obtain results for this experiment, k-folds cross validation can be used (Kohavi, 1995). For each theory, the proofs within are partitioned into k non-overlapping sets. SEPIA invokes MINT (Walkinshaw, Taylor, and Derrick, 2015) to infer a model from $k-1$ sets of proofs, and uses the model to try and prove the theorems within the remaining set. This process repeats until each set has been used exactly once as the set of theorems to be proven.

As described in Chapter 4, some aspects of the proof search can be configured. For this evaluation, the inferred model is allowed to be searched for a maximum of 60 seconds. This is a reasonable period of time, and a realistic amount of time that a user might be willing to wait for a result. The type of search performed is a breadth-first search. This was chosen so that SEPIA proposes shorter proofs contained within the models.

This evaluation uses the leave-one-out variant of cross validation. This is equivalent to setting k to be the number of theorems contained within the theory (as the dataset is split into k partitions). This has the effect that all other theorems within a theory are used to infer a model from. This mirrors the standard implementation of SEPIA, where all of the theorems the user selected are used during model inference.

The results obtained from this result will demonstrate the potential for SEPIA to be used for automatic proof generation. The success of this experiment will also justify whether state-machine inference is a potentially useful technique in the context of theorem proving. Additionally, when Coq-PR^3 is evaluated, the results from the corresponding experiment can be compared to see whether Coq-PR^3 improves upon SEPIA.

6.2.3 Comparing with existing Coq automation

Coq provides a number of automated tactics and decision procedures that may be invoked during a proof attempt (Bertot and Castéran, 2004). Although not designed to prove theorems that are highly complex, they can nevertheless discharge some trivial statements. Also, if a problem falls into a specific class of problems (e.g. propositional

logic), then these tactics may be able to fully discharge the theorem. The main tactics that are considered in this evaluation are:

- **auto/eauto:** performs simple proof search, applying previously defined facts. However, they both use a very small subset of Coq's tactics.
- **tauto:** a propositional tautology solver that can discharge goals that fall within this class.
- **trivial:** a limited version of **auto**, designed to handle subgoals that could contain equalities.
- **firstorder:** an experimental tautology solver for subgoals containing first-order inductive definitions.

Many of these tactics can be controlled with the help of additional parameters. Hint databases are collections of previously defined facts (theorems, definitions) that are grouped together by domain (e.g. sets). The user can specify which hint databases to provide to these automated tactics. Initially, the tools only invoke a basic core of facts. In this evaluation, where hint databases can be specified, the automated tactics are allowed access to all available hint databases, plus the theorems that SEPIA infers a model from.

Some of the automated tactics in Coq may also be specified with a search depth. The default search depth (set to 5) is used throughout these experiments. Another factor to consider is the time allowed for the automated tactics to prove the theorem. Coq provides a `timeout` command that allows a limit to be specified. As standard throughout this chapter, a timeout of 60 seconds is used for both the automated tactics and SEPIA/Coq-PR^3 proof search algorithms.

6.2.4 Properties of discovered proofs

So far, the experiments described capture the number of theorems proven by SEPIA. Although this provides a lot of information about the effectiveness of the technique, it doesn't examine the proofs that SEPIA produces. This evaluation delves deeper and examines the proofs that were discovered automatically by SEPIA. Specifically, any interesting properties about the automated proofs are investigated.

The evaluation uses Coq datasets that have already been completed by a team of experts. Therefore, there is access to the hand-curated proofs that have been completed. One simple consideration is to analyse the length of the SEPIA proof compared to

the manually crafted proof. Specifically, did SEPIA discover a shorter proof than the existing manually created one.

The second aspect of this experiment looks at the discovered proofs in relation to the proofs used to infer the model. In Chapter 2, the process of state-merging was described. The main result of using these algorithms is that the state machines produced are a generalisation of the input sequences. Crucially this means that in addition to allowing all tactic sequences that were available initially, the model may well suggest unseen sequences of tactics that weren't present in the initial traces. In this thesis, these are referred to as "new" tactic sequences.

6.2.5 Measuring the success of Coq-PR^3

As with the SEPIA experiment, this evaluation is performed on a per-theory basis. For each theorem, the similar ones are obtained from the proof clustering algorithms. Then, a model is inferred from the suggestions produced by ML4PG instead of using all theorems. Again, the resulting model is searched automatically for 60 seconds using breadth-first search.

An additional parameter used in this experiment is the *granularity* setting in ML4PG. This is used to control the size and precision of the clusters generated. Using $g = 1$ leads to a few large and general clusters being produced, whereas $g = 5$ generates many smaller and more precise clusters. To see if varying this parameter has any effect on the overall technique, the granularities utilised are $g = \{1,3,5\}$.

As with SEPIA, the proofs that Coq-PR^3 produces are evaluated based on whether they are shorter than the human generated proof, or if they contain a new sequence of tactics. The Coq-PR^3 approach was designed to enhance the original SEPIA approach. Therefore, the number of additional proofs that Coq-PR^3 produces are presented. Ideally, to show that Coq-PR^3 is a useful technique it should prove theorems that weren't proven by SEPIA. Also, the technique should retain the overall benefits of SEPIA such as using new sequences of tactics and identifying shorter proofs.

6.3 Results

The rest of this chapter focusses on presenting the results from running the experiments. These experiments provide potential answers to the research questions defined at the start of this chapter. By the end of this evaluation, the overall effectiveness of SEPIA and Coq-PR^3 will become clearer.

6.3.1 RQ1: Does SEPIA prove Coq theorems automatically?

Table 6.1 presents the results for the experiment described in Section 6.2.2. The library name and the size (number of theorems) are displayed under the Library and Size columns respectively. The column headed SEPIA shows the number of theorems that SEPIA proved automatically.

TABLE 6.1: Number of theorems proven by SEPIA

Library	Size	SEPIA
Ssreflect	1389	$430 \approx 31\%$
CompCert	3248	$420 \approx 13\%$
Total	**4637**	**850 ≈18%**

Looking at each dataset as a whole in Table 6.1, 31% (430 out of 1389) of the theorems in Ssreflect were proven with SEPIA. In CompCert, SEPIA managed to prove 13% (420 out of 3248) completely automatically using SEPIA. Overall, on all of the theorems evaluated, using SEPIA yielded a proof for 850 of them – corresponding to a success rate of 18%.

This experiment was run using individual theories – therefore the results will naturally differ dramatically on a per-theory basis. In CompCert, there were 19 theories where SEPIA failed to find a single proof. On investigation, this can be explained by various factors. Firstly, many of these theories contained less than 10 theorems within them. Therefore, there is a small selection of data available to infer a model from. A second reason is that in some of the larger theories, the search was still in progress when the 60 second limit was reached.

In Ssreflect, there were no theories where SEPIA was unable to find a proof. The lowest success rate within an individual theory was 18% in the `fintype` theory. Apart from 6 proof attempts that exhaustively searched the complete automaton, all of the other proof attempts were still in progress when the 60 second timeout was reached.

The highest success rates within individual theories were between 40% and 50% in both datasets. In the `ssrbool` theory from Ssreflect, nearly 50% of the theorems were proved by SEPIA. This is partly due to the relatively simple subject matter contained within the theory. In CompCert, SEPIA proved 43% of the theorems in the `Ordered` theory – where many proofs were discharged using the same sequence of tactics.

Another interesting aspect of the proof attempts is to look at the overall time needed to find a proof. This takes into account the model inference time plus the (up to) 60 seconds of proof search. In Ssreflect, the longest time needed to find a proof was 1 minute 15 seconds, and the shortest was under 1 second. In CompCert, the longest

time to find a proof was 40 minutes and 9 seconds. This was in the `Memory` theory, that contained over 300 theorems, many containing long proofs. The model inference algorithms required a long time to infer a model from the proofs within this theory. The shortest time needed to find a proof in CompCert was again under a second.

The average time needed to identify a proof can be calculated from adding the model inference time and the (up to) 60 seconds used to search the model. The average proof attempt time in Ssreflect was just under 2 seconds. This is helped by the relatively small pool of tactics used in Ssreflect, and the typically short proofs. In CompCert, on average a proof was found after 44 seconds of effort (model inference plus search). This increase in time over Ssreflect can be explained by CompCert being a much more diverse library of proofs, with little common reasoning patterns and a larger pool of tactics used.

The results from this experiment show that SEPIA is a promising technique for proof automation in Coq. Given a set of proofs, SEPIA is able to infer a model, and use this as a basis for automated proof generation. The models contain sequences of tactics to apply in Coq. In many cases the overall process is reasonably quick at obtaining a proof. In any proof development, being able to prove 18% of the theorems automatically signifies a large reduction in the amount of human effort required.

6.3.2 RQ2: Is SEPIA more effective than existing Coq automation?

Table 6.2 shows the results for the experiment defined in Section 6.2.3. For each proof development, the name and number of theorems is in the column headed Library and SIze respectively. The automated Coq tactics used in this experiment are grouped together under the column Coq-Tactics.

TABLE 6.2: Comparison of SEPIA and various automated Coq tactics

Library	Size	SEPIA	Coq-Tactics
Ssreflect	1389	430	155
CompCert	3248	420	120
Total	**4637**	**850≈18%**	**275 ≈6%**

In the CompCert development, the built-in Coq tactics managed to prove 120 theorems. This corresponds to a success rate of approximately 4% across the whole development. In Ssreflect, the tactics managed to prove around 11% of the theorems. Overall, using all of the theorems used in this evaluation, the automated tactics proved 275 Coq theorems, leading to a success rate of just over 6%.

In comparison, SEPIA managed to prove nearly three times as many theorems in SSreflect and nearly four times as many in CompCert. Overall, 18% of the theorems within this evaluation has a proof discovered by SEPIA. This represents a significant gain over existing automation within Coq. These results are not entirely surprising – as already established the Coq tactics are typically limited in which tactics they can try. In comparison, SEPIA is able to try a larger pool of tactics as *any* set of proofs could be input into the model inference.

An interesting question not answered by the raw data is the overlap between SEPIA and the Coq automation. For instance, are there theorems that the automated tools proved but SEPIA was unable to find a proof. Conversely, how many proofs did SEPIA derive that the Coq tactics didn't. Table 6.3 takes the data from Table 6.2 and shows the number of proofs found exclusively by either technique, plus the number of theorems proven by both.

In Table 6.3, the number of proofs that SEPIA exclusively discovered are shown in the column headed SEPIA. Similarly, the number of proofs discovered only by the automated tactics are under the column named Coq-Tactics. Finally, the number of proofs discovered by both techniques are in the final column headed Both.

TABLE 6.3: Proofs found per-technique

Library	SEPIA	Coq-Tactics	Both
Ssreflect	306	31	124
CompCert	324	24	96
Total	**630**	**55**	**220**

In Ssreflect, the Coq tactics managed to prove 31 theorems that SEPIA was unable to prove, and 24 in CompCert. The number of theorems proved by both techniques totalled 220 across both datasets – 124 in Ssreflect and 96 in CompCert. This means that when disregarding the theorems that both techniques proved, SEPIA automatically proves 630 theorems that weren't proven by existing techniques.

These results demonstrate that SEPIA enhances the existing proof automation available within Coq. There are situations where SEPIA may be unable to prove a theorem, but the automated tactics can. This demonstrates that the user should invoke existing automation before calling SEPIA. However, there are a vast majority of cases (nearly 75%) where SEPIA discovers a proof when the existing automation couldn't.

6.3.3 RQ3: Are there "interesting" properties of the proofs discovered?

The next experiment studies the actual proofs that SEPIA discovered, using the methodology described in Section 6.2.4. Specifically, the experiment attempts to draw some comparisons between the SEPIA generated proofs and the original hand-curated proofs available within the Ssreflect and CompCert proof developments. Alama *et al.* have completed an evaluation of ATP proofs, comparing them to human created proofs (Alama, Kühlwein, and Urban, 2012). The main observation was that machines can generally find simpler proofs than humans – for instance use fewer steps, or need fewer dependencies.

There are two sets of criteria that this evaluation uses to compare the proofs. Firstly, the length of the human and automated proofs are compared to see if the automated proofs are shorter. Secondly, the discovered proof is evaluated against the theory used to infer the model. The proof steps are checked to see if they were present within an existing proof, or SEPIA has proposed a "new" sequence of tactics. The experiments used SEPIA in breadth-first search mode – so it is reasonable to expect that the proofs found may be shorter than human derivations. If there is a large proportion of new tactic sequences, then this justifies the choice of state-machine inference.

In Ssreflect, 26 of the proofs that SEPIA discovered were shorter than the corresponding human derivation. The Ssreflect set of tactics are designed to help write shorter proofs, so the chances of identifying shorter proofs in this dataset is limited. It is more than likely the case than many of the proofs are already as short as they can be, thanks to the design of the language and the expertise of the user who completed the proofs.

In CompCert, SEPIA managed to discover a much larger proportion of shorter proofs. Out of the 420 proofs that SEPIA found, 238 of them were shorter than the corresponding human proof, and 38 were longer. This corresponds to just over 55% of the proofs being shorter. To explain this, it is likely that SEPIA had access to a diverse set of proofs that it was able to use to produce the shorter derivations. It may also be the case that the CompCert development wasn't refactored at all. Also, the combination of state merging algorithms and a breadth-first search means SEPIA was able to identify these shorter proofs.

The next aspect of this experiment was to look at the number of new proofs found using SEPIA. For each proof SEPIA discovered, it was compared to the proofs that were used to infer the model. A new proof is defined as one that contains a sequence of tactics that wasn't contained within the original set of proofs. This usually means that the discovered tactic sequence is pieced together from fragments of different proofs.

Otherwise, a proof is "reused" in the sense that the discovered tactic sequence can be found within the proof library.

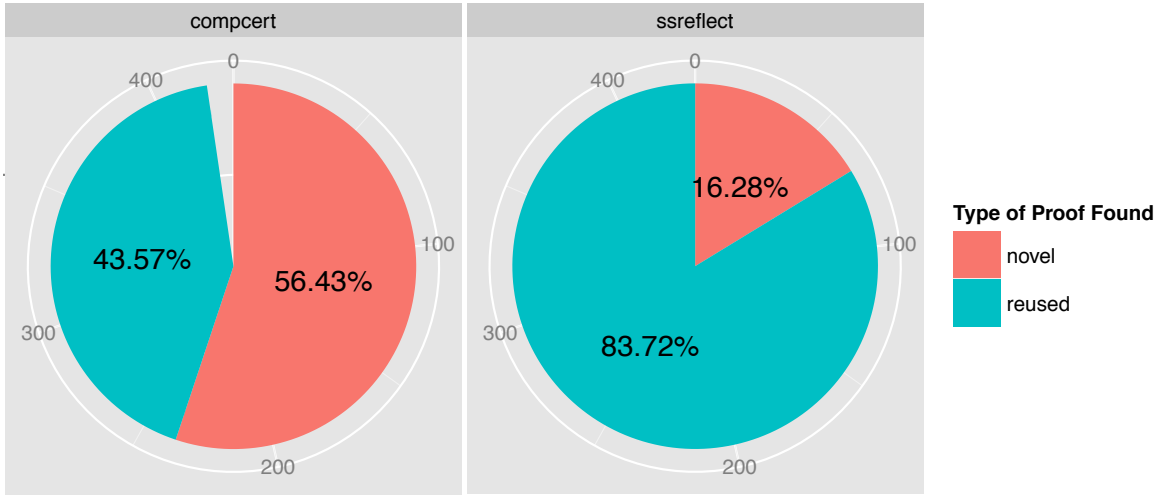


FIGURE 6.1: Comparing new and reused proofs

For each dataset, Figure 6.1 shows the proportion of new and reused proofs. The results show that SEPIA is able to reuse existing proofs effectively, but also identify many new combinations of tactics. This benefit further backs up the potential of using EFSMs in theorem proving.

In CompCert, over 50% of the proofs found by SEPIA constituted new combinations of tactics. The diverse nature of the development means there are many cases where there are no common reasoning patterns, and many different tactic sequences are used. These new tactic sequences represent examples where there were links between different proofs that were identified during model inference.

In Ssreflect, there were a lot more re-used tactic sequences, and less than 20% of the proofs discovered constituted new sequences. This is partly due to the fact that the Ssreflect theories prove theorems about more general concepts such as sequences and natural numbers. Many of these proofs will typically use the same common reasoning pattern. Therefore, there is less diversity within the corpus and less chance of combining steps from different proofs.

Potential proof refactoring application The results from this section suggest an interesting application of SEPIA – proof refactoring. This experiment has shown that it is possible to automatically run SEPIA on a theory and get a list of suggestions regarding shorter and new derivations. This can be thought of as a way of refactoring proof scripts. Refactoring is a popular technique in software engineering and can be described as changing code without changing the functionality. In the case of interactive proofs, this loosely equates to changing the proof without changing the conjecture.

Proof refactoring is an area that has received moderate attention, although recently the field has become more aware of the idea. Aspinall and Kaliszyk defined the idea of formal proof metrics (Aspinall and Kaliszyk, 2016). These are a set of measurements that are used to monitor and compare proof developments. They start by looking at common object-oriented programming metrics and define corresponding ones for formal proofs. This work forms a useful baseline for further exploratory work into this area.

Another approach defined by Whiteside *et al.* is proof-script refactorings (Whiteside et al., 2011). By again drawing comparisons to software engineering, it is possible to define refactorings that can preserve the semantics of a proof script but improve its readability and maintainability. Examples of possible refactorings include simple operations such as rename a lemma, through to more complex ones that safely delete unused theorems and facts.

The two approaches mentioned here demonstrate that there is a use in theorem proving for automated tools inspired by software engineering methods. SEPIA also has a place within this set of tools by providing means of analysing the existing proofs and suggesting other derivations. A possible mode (in addition to the proof mode) could be for SEPIA to automatically suggest a list of changes that can be reviewed manually.

6.3.4 RQ4: Does Coq-PR^3 improve upon SEPIA?

The final experiment (defined in Section 6.2.5) studies the Coq-PR^3 proof pattern recycling approach described in Chapter 5. Table 6.4 shows the number of theorems that were proven by each configuration of Coq-PR^3 – *g* stands for granularity in the column headers. Results for each value of granularity used are shown in their respective column.

TABLE 6.4: Number of theorems proven by Coq-PR^3

Library	Size	Coq-PR^3 (g=1)	Coq-PR^3 (g=3)	Coq-PR^3 (g=5)
Ssreflect	1389	329	336	335
CompCert	3248	247	249	246
Total	**4637**	**576**	**585**	**581**

In terms of the number of proofs found, Table 6.4 shows that all configurations of Coq-PR^3 lead to proofs being discovered automatically. Interestingly, using Coq-PR^3 leads to fewer theorems being proven than when using SEPIA (see Table 6.1). To a degree this is to be expected – Coq-PR^3 restricts the number of theorems that a model is inferred from. However, it could be that the proof clustering algorithms used in ML4PG

don't produce optimal clusters of Coq proofs. However, the true measure of success of Coq-PR^3 is the number of additional proofs that were discovered using the technique – compared to using SEPIA.

Table 6.5 shows the number of additional proofs that Coq-PR^3 found (i.e theorems that SEPIA didn't prove in earlier experiments). Each configuration of granularity is shown in its respective column The final column headed provides the number of additional proofs discovered when using any of the granularity choices.

TABLE 6.5: Number of additional proofs discovered by Coq-PR^3

Library	Coq-PR^3 (g=1)	Coq-PR^3 (g=3)	Coq-PR^3 (g=5)	Any
Ssreflect	28	26	26	30
CompCert	78	72	92	141
Total	**106**	**98**	**118**	**171**

For Ssreflect, using $g = 1$ led to 28 additional theorems being proven that SEPIA was unable to. Using $g = 3$ led to 26 being proven, whilst $g = 5$ also led to 26 additional proofs. Of course, these is some overlap between these results, there were 30 theorems in total (across all granularities) that Coq-PR^3 proved that SEPIA was unable to.

Similarly, in CompCert $g = 1$ led to 78 additional theorems being proven. The granularity setting $g = 3$ led to 72 additional proofs, whilst $g = 5$ led to 92 additional proofs being discovered. Disregarding the overlap in these examples means that 141 additional theorems were proven using Coq-PR^3. Between the two datasets used, Coq-PR3 discovered an additional 171 proofs, meaning an extra 3% of the theorems were proven using Coq-PR^3.

When compared to SEPIA, the average time taken to find a proof (model inference time plus search) was under 2 seconds for all choices of granularity. This is similar to the performance of SEPIA. In CompCert, using $g = 1$ meant that a proof was found on average in around 5 seconds. Using $g = 3$ led to a proof being discovered after 3 seconds of effort. Finally, $g = 5$ led to a proof being found after 2 seconds on average. These results demonstrate that Coq-PR^3 will generally find a proof in much less time than SEPIA. This is due to the time reduction in both model inference and the search of the reduced model.

There is no clear setting of granularity that produces the "best" results. Although some theorems are proven by all three configurations, each setting has some theorems that were uniquely proven only by that particular granularity. In practice, it would be advisable for a user to invoke Coq-PR^3 using the smallest clusters first, then change the granularity when needed.

The additional proofs that Coq-PR^3 discovered were also compared to the ones found by the automated Coq tactics (see the earlier experiment). In CompCert, of the 141 additional theorems proven by Coq-PR^3, only 2 were able to be proven by the Coq tactics. In Ssreflect, all 30 additional proofs were ones that weren't proven by the Coq tactics. These results are encouraging, and demonstrate the additional benefits that Coq-PR^3 brings. Furthermore, Coq-PR^3 and SEPIA both provide a complement to the automation already available within Coq.

The final aspect of this experiment is to ensure that Coq-PR^3 retains the main benefits of SEPIA. The proofs are compared in the same manner as earlier. They are checked for new tactic sequences, and whether they are shorter than the corresponding human derivations. Table 6.6 shows the number of new and shorter proofs (when compared to the library proofs) found per granularity configuration.

TABLE 6.6: Shorter proofs and new tactic sequences discovered by Coq-PR^3

Library	Coq-PR^3 (g=1)	Coq-PR^3 (g=3)	Coq-PR^3 (g=5)
Ssreflect	17 shorter 6 new	16 shorter 4 new	16 shorter 5 new
CompCert	86 shorter 197 new	81 shorter 187 new	45 shorter 199 new

In Ssfreflect, these proportions of new and shorter proofs matches what was seen in earlier experiments. There is limited chance of finding shorter proofs due to the nature of the tactic language, and the subject matter of the proofs. In CompCert, the results really demonstrate that Coq-PR^3 preserves the useful benefits of the original SEPIA technique. A large proportion of the proofs found using Coq-PR^3 consisted of new tactic sequences, whilst roughly a quarter of the Coq-PR^3 proofs were shorter than the manually created derivations.

The results from this experiment are encouraging, and demonstrate that the Coq-PR^3 technique does enhance SEPIA. The main aim of Coq-PR^3 is to prove theorems that may not be proven by SEPIA due to the underlying design choices. An additional 171 theorems were proven using the methods provided within Coq-PR^3 – meaning that between the two tools presented in this thesis, 22% of the theorems within the evaluation have been proved automatically.

6.4 Conclusions

This section defined a set of research questions and experiments that were used to evaluate SEPIA and Coq-PR^3. Using two large and diverse Coq datasets (Ssreflect and CompCert), the two tools were fully evaluated to assess their potential for automatic proof generation in Coq. The results obtained within this chapter are particularly encouraging.

SEPIA is an approach that uses state-machine inference techniques to generate state-machine models from Coq proof corpora. These state machines are useful as they can transform a proof library into a more concise representation. Furthermore, the models can be searched automatically in an attempt to formulate proofs. These models have been shown to be reasonably accurate, as they are able to prove 18% of the theorems used in this evaluation.

Coq provides some simple automation in the form of tactics and decision procedures. SEPIA was compared to these tools to see how many theorems they can prove. These tactics are typically limited by the possible steps that they try, and are more suited to discharging trivial subgoals. The evaluation demonstrated that SEPIA could discharge a large number of theorems that these automated tactics couldn't. The overlap between the techniques was considered, and it was shown that SEPIA provides a complementary technique to what is already provided in Coq.

The proofs that SEPIA discovered were also evaluated within this chapter. There were two main "interesting" properties considered. Firstly, was SEPIA able to identify shorter proofs than the corresponding human derivations. Secondly, did SEPIA propose a new sequence of tactics that wasn't present within the corpus, or did it simply reuse existing proofs. The results demonstrated that SEPIA produced a shorter proof in just under a third of cases. Similarly, just over a third of the proofs that SEPIA discovered were new sequences of Coq tactics.

The final experiment looked at the additional benefits that Coq-PR^3 brings over using SEPIA alone. By enhancing SEPIA with additional machine learning, an additional 171 theorems were proven. This justifies the choices made when designing Coq-PR^3. Crucially, the benefits that SEPIA brings (such as shorter and new proofs) are retained within Coq-PR^3. All but two of the theorems that Coq-PR^3 proved were outside the scope of the existing automated tools within Coq.

Overall, the results presented within this evaluation demonstrate the potential of SEPIA and Coq-PR^3. The aim was to show that SEPIA is able to prove theorems in Coq, and that the enhancements provided by Coq-PR^3 can improve the results further. In addition, several additional benefits of using the tools have been shown – for instance the inferred models produced by SEPIA proposing completely new sequences of

tactics.

Chapter 7

Conclusions and Future Work

This chapter presents the conclusions and future work arising from the techniques presented within this thesis. Section 7.1 summarises the research and its outcomes. Section 7.2 discusses the conclusions drawn from the presented research. Finally, Section 7.3 described the various avenues of future work.

7.1 Overall summary

The work in this thesis introduces new techniques that combine learning and theorem proving. Firstly, the SEPIA technique is described, that uses state-machine inference techniques to generate models from Coq proofs. These models summarise the tactic applications present within a corpus of proofs. The inferred models can then be used manually or automatically to generate proofs. Then, additional machine learning is used to form Coq-PR^3 – a suite of intelligent tools for Coq. To summarise, each chapter contained the following work:

- Chapter 2 introduced the areas of theorem proving and machine learning. Previous combinations of these areas were summarised. Then, the area of state-machine inference was described in detail, as it forms the basis of the rest of the work within this thesis.

- Chapter 3 covered the first stage of the SEPIA process. This involves taking Coq proofs and turning them into a format that can be used by model inference algorithms. The MINT model inference technique (Walkinshaw, Taylor, and Derrick, 2015) is then used to infer models from the proof traces. Finally, some case studies demonstrated how the models could be used manually to derive proofs.

- Chaper 4 introduced an algorithm that can search the inferred models as prove theorems automatically in Coq. This algorithm forms the basis of a plugin for

ProofGeneral (Aspinall, 2000) that allows SEPIA to be invoked during proof attempts. The plugin is demonstrated with an example – showing the complete process of using SEPIA to prove a Coq theorem.

- Chapter 5 studied potential improvements to the SEPIA technique by adding additional machine learning. The ML4PG proof clustering technique (Komendantskaya, Heras, and Grov, 2013) is combined with SEPIA to form Coq-PR^3 – an intelligent suite of tools that can be used within ProofGeneral. The resulting combination addresses some limitations of the original SEPIA technique.

- Chapter 6 presented the evaluation of SEPIA and Coq-PR^3. The results indicated that SEPIA and Coq-PR^3 are useful tools that are able to prove theorems automatically in Coq. They also complement existing Coq automated tactics by proving a larger selection of theorems than before.

7.2 Conclusions

The potential uses of proof assistants such as Coq span both mathematics and industry. Despite their championing by computer scientists and mathematicians, they remain a tool used in niche scenarios instead of being applied more widely. The large proof developments to date have shown that although practical, there is a heavy reliance on human interaction to complete the proofs manually.

This thesis set out to investigate the potential use of state-machine inference to infer models from interactive proofs. The overall aim was to use information from existing proofs as a basis to automate further proof attempts. The work completed led to the creation of a set of techniques that combine theorem proving and machine learning for the purposes of proof exploration and automation.

The first technique – SEPIA – allows proof developers to select a collection of Coq proofs, and obtain a descriptive model of the tactics used. These models can then be used manually to help the user formulate a proof in Coq. Additionally, thanks to a ProofGeneral extension, SEPIA is able to communicate with Coq. This enables the automatic search of the inferred models, meaning that proof attempts can be automated using the inferred model as a search space.

The proofs discovered using SEPIA are interesting for two reasons. Firstly, when compared to human derivations the automated proofs may typically be shorter. Secondly, the underlying model inference algorithms produce state-machines that contain tactic sequences that weren't present in the initial corpus of proofs. These new sequences were shown to be useful in proving theorems in Coq.

The second technique – Coq-PR^3 – incorporates additional learning using proof clustering techniques from ML4PG. By doing this, some of the limitations in the original SEPIA technique are addressed. Coq-PR^3 has also been implemented as a Proof-General extension, providing the user a full suite of machine learning tools for Coq. The suite allows the user to invoke SEPIA and ML4PG as individual tools, or as a combined proof search algorithm that uses the benefits of both tools.

The evaluation demonstrated SEPIA and Coq-PR^3 on two large Coq proof developments – Ssreflect (Gonthier, 2005) and CompCert (Leroy, 2009). The results highlighted numerous benefits of the approaches. The new techniques can prove more theorems than existing Coq automated tactics. In addition, the experiments showed that the Coq-PR^3 technique can bring additional benefits over using SEPIA on its own.

The novelty of the approaches, combined with the evaluation results has suggested a large amount of future work. The various components of SEPIA and Coq-PR^3 can all be explored further. In terms of the state machine inference, there are interesting avenues for including negative data and designing custom state-merging algorithms. To improve the theorem proving side, various enhancements could be made by making the search procedure more intelligent or investigating the addition of more machine learning.

Overall, this thesis has introduced two useful tools to help prove theorems automatically in Coq. They enhance the level of proof automation available in Coq. Aside from providing the initial set of theories, the approaches work completely automatically – taking the theories, learning from them and applying the learned knowledge during proof attempts. By improving the level of proof automation, theorem provers such as Coq might one day be used more widely to produce dependable IT systems. Also, in mathematics new discoveries could be found in the future using proof assistants.

7.3 Future Work

This thesis has led to the creation of SEPIA and Coq-PR^3 – these are approaches to automate proofs in Coq using machine learning techniques. The novelty of this work means that there are many opportunities for future work. In conjunction with the evaluation in Chapter 6 the main areas for improvement and further research are described below.

Searching the models Chapter 3 demonstrated that the inferred state machines are stored as directed graphs. This means that standard graph traversal algorithms can be used to search the model during a proof attempt. However, when dealing with diverse proof corpora the inferred models invariably become extremely large. Standard graph

traversals will still exhaustively search the model regardless of size, however there may be more intelligent ways of searching a model. Heuristic search techniques (Pearl, 1984; Russell and Norvig, 2010) can help in scenarios such as these.

One aspect that could be harnessed whilst searching the model is information about the proof state at a given point. The proof search interacts with Coq, but currently only logs if the tactic made progress or not. By incorporating other information into the search, more informed choices about which paths through the model are most promising. Such information could include the number of subgoals at a given point, amount of information in the context or the overall structure of the current subgoal.

Premise selection for SEPIA Given a library of proofs, SEPIA will be invoked using all of the exemplar proofs available. This can lead to a large state machine being inferred, with a large amount of redundant information being captured. The main avenue of future work here is to look at techniques for identifying fruitful examples. We have already shown how Coq-PR^3 combines SEPIA with proof clustering approaches. However, there are library search mechanisms and dependency analysis (Alama, Mamane, and Urban, 2012; Kaliszyk, Mamane, and Urban, 2014) techniques that may also provide useful suggestions.

Domain specific state merging In Chapter 3, the state merging process is described. Currently, off-the-shelf algorithms are applied and are successful. Another direction of future work could be to look at how state-merging algorithms work and devise an approach that is specifically for the domain of interactive proofs. States are merged based on their outgoing behaviour (in the current case these are the subsequent tactics applied) – it may be possible to incorporate extra information from the proof state into this process.

Incorporating negative data Proof attempts generate a lot of failed derivations – tactics that were tried but unsuccessful, or sequences of tactics that led to a dead end. One possible avenue would be to incorporate this information back into the model inference process (Walkinshaw, Derrick, and Guo, 2009). Proof traces can be grouped by whether they were successful or not. By incorporating this into the model inference, the models could become more accurate, as they will not allow sequences of tactics that were unsuccessful. A well known result from Gold suggests that the inference of languages is greatly helped with the inclusion of negative examples (Gold, 1967).

Experimenting with different classifiers MINT is connected with Weka, a suite of over 100 machine learning algorithms. For all of the experiments and work completed

so far, the J48 decision tree learner has been used, as the early experiments were encouraging. However, it may be the case that other classifiers may perform better in the domain of theorem proving. Additionally, Weka allows custom classifiers to be created. An interesting avenue of future work could be investigating this idea further. Some changes would been to be made so that the inferred models can be annotated with the output from other classifiers. Additionally, there are many features of interactive proofs that could be used to create a custom classifier for the theorem proving domain.

Application to other theorem provers In this work, Coq has been the main focus owing to its popularity but also the lack of effective automation available. Other theorem provers such as Isabelle (Nipkow, Wenzel, and Paulson, 2002) provide stronger automation. One possible avenue of work would be to compare SEPIA with more established tools such as Sledgehammer (Meng, Quigley, and Paulson, 2006). The SEPIA approach is generic and is (in principle) applicable to any tactical theorem prover. The actual model inference task remains the same, regardless of the theorem proving language. However, to extend SEPIA there are 2 problems regarding input/output that must be overcome. The first is how to convert the raw proof script into a proof trace. Secondly, SEPIA must be able to interact with a running instance of the theorem prover.

One challenging aspect of applying SEPIA to other provers is the input language (Harrison, 1996) of the theorem prover. They can be grouped into *procedural* and *declarative*. Procedural input is slightly easier to use as it typically constitutes the application of proof tactics. Declarative languages are representative of natural language, where it is much harder to elicit the underlying proof steps used due to the extra noise involved. One theorem prover that is similar to Coq is Lean (de Moura et al., 2015) – this could be a useful theorem prover to port the SEPIA technique to as it uses a similar syntax.

Using SEPIA for proof refactoring The experiments in Chapter 6 showed that the techniques were effective at taking a collection of proofs and identifying shorter or different derivations. The idea of automatically refactoring proofs is an interesting one, and has so far had moderate attention. Whiteside *et al.* propose a series of proof script refactorings (Whiteside et al., 2011) that aim to tidy up the structure of interactive proof scripts. However, they do not actually make changes to the proofs themselves. Aspinall and Kaliszyk have also investigated various proof metrics (Aspinall and Kaliszyk, 2016). These metrics can be used to compare proof developments or investigate refactorings. SEPIA could be deployed in a theory analysis mode that provides automated suggestions for shorter and different proofs that the proof developer could then use if required.

Appendix A

Source Code Listings

This Appendix contains source code listings for various parts of the SEPIA implementation.

A.1 `NodeInformation` class

Listing A.1 shows the code for the `NodeInformation` class. This was described in Chapter 4, and is used to keep track of information during the proof search.

A.2 Sample `NodeInformation` comparator

The code in Listing A.2 shows a potential comparator that can be used during heuristic search. In this case, it compared two `NodeInformation` objects and sorts them based on the number of subgoals.

LISTING A.2: Simple SEPIA heuristic

```
public class SimpleSubgoalHeuristic implements
Comparator<NodeInformation> {

        @Override
        public int compare(NodeInformation o1,
        NodeInformation o2) {
                if (o1.subgoals < o2.subgoals) return -1;

                if (o1.subgoals > o2.subgoals)  return 1;

                return 0;
        }
}
```

LISTING A.1: NodeInformation class in SEPIA

```
public class NodeInformation {

//State to visit next in the machine
private Integer node;

//The incoming path (if any)
private LinkedList<String> incomingPath;

//Default constructor
public NodeInformation(){
        incomingPath = new LinkedList<String>();
}

public void setIncomingPath(LinkedList<String> ip){
        incomingPath.addAll(ip);
}

public void setNode(Integer n){
        node = n;
}

public Integer getNode(){
        return node;
}

public LinkedList<String> getIncomingPath(){
        return incomingPath;
}
```

Bibliography

Alama, Jesse, Daniel Kühlwein, and Josef Urban (2012). "Automated and Human Proofs in General Mathematics: An Initial Comparison". In: *Logic for Programming, Artificial Intelligence, and Reasoning: 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*. Ed. by Nikolaj Bjørner and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 37–45.

Alama, Jesse, Lionel Mamane, and Josef Urban (2012). "Dependencies in Formal Mathematics: Applications and Extraction for Coq and Mizar". In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring et al. Vol. 7362. Lecture Notes in Computer Science. Springer, pp. 1–16.

Aspinall, David (2000). "Proof General: A Generic Tool for Proof Development". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Susanne Graf and Michael Schwartzbach. Vol. 1785. Lecture Notes in Computer Science. Springer, pp. 38–43.

Aspinall, David and Cezary Kaliszyk (2016). "Fundamental Approaches to Software Engineering: 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings". In: ed. by Perdita Stevens and Andrzej Wąsowski. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Towards Formal Proof Metrics, pp. 325–341.

Bertot, Yves and Pierre Castéran (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer.

Blanchette, Jasmin et al. (2015). "Mining the Archive of Formal Proofs". In: *Conference on Intelligent Computer Mathematics (CICM 2015)*. Ed. by M. Kerber. Vol. 9150. Invited paper, pp. 3–17.

Blanchette, Jasmin et al. (2016). "Hammering towards QED". In: *Journal of Formalized Reasoning* 9.1, pp. 101–148. ISSN: 1972-5787.

Blanchette, Jasmin Christian et al. (2012). "More spass with isabelle". In: *International Conference on Interactive Theorem Proving*. Springer, pp. 345–360.

Böhme, Sascha and Tobias Nipkow (2010). "Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings". In: ed. by Jürgen Giesl and Reiner Hähnle. Springer Berlin Heidelberg. Chap. Sledgehammer: Judgement Day, pp. 107–121.

Bridge, James P., Sean B. Holden, and Lawrence C. Paulson (2014). "Machine Learning for First-Order Theorem Proving". In: *Journal of Automated Reasoning* 53.2, pp. 141–172.

Bundy, Alan (1988). "The use of explicit plans to guide inductive proofs". In: *9th International Conference on Automated Deduction: Argonne, Illinois, USA, May 23–26, 1988 Proceedings*. Ed. by Ewing Lusk and Ross Overbeek. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 111–120.

Carlson, A. et al. (1999). *The SNoW Learning Architecture*. URL: `http://cogcomp.cs.illinois.edu/papers/CCRR99.pdf`.

Cheng, Kwang Ting and A. S. Krishnakumar (1993). "Automatic Functional Test Generation Using the Extended Finite State Machine Model". In: *Proceedings of the 30th International Design Automation Conference*. DAC '93. Dallas, Texas, USA: ACM, pp. 86–91.

Chlipala, Adam (2011). *Certified Programming with Dependent Types*. `http://adam.chlipala.net/cpdt/`. MIT Press. URL: {`http://adam.chlipala.net/cpdt/`}.

de Moura, Leonardo et al. (2015). "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, pp. 378–388.

Delahaye, David (2000). "A Tactic Language for the System Coq". In: *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*. LPAR'00. Reunion Island, France: Springer-Verlag, pp. 85–95. ISBN: 3-540-41285-9.

Duncan, Hazel (2008). "The use of data mining for the automated formation of tactics". PhD thesis. University of Edinburgh.

Ernst, Michael D. et al. (2007). "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* 69.1–3. Special issue on Experimental Software and Toolkits, pp. 35 –45.

Gold, E. Mark (1967). "Language identification in the limit". In: *Information and Control* 10.5, pp. 447–474.

Gonthier, Georges (2005). *A computer-checked proof of the four colour theorem*.

— (2008). "Formal proof–the four-color theorem". In: *Notices of the AMS* 55.11, pp. 1382–1393.

Gonthier, Georges et al. (2013). "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, pp. 163–179.

Gordon, Michael J. C., Robin Milner, and Christopher Wadsworth (1979). "Edinburgh LCF: A Mechanized Logic of Computation". In: *Lecture Notes in Computer Science 78*. Springer-Verlag.

Gransden, Thomas (2013). "Boosting Automated Reasoning by Mining Existing Proofs". In: *20th Automated Reasoning Workshop (ARW)*.

— (2015). "Combining ML4PG and SEPIA". In: *22nd Automated Reasoning Workshop (ARW)*.

Gransden, Thomas, Neil Walkinshaw, and Rajeev Raman (2014). "Mining State-Based Models from Proof Corpora". In: *Intelligent Computer Mathematics*. Ed. by Stephen M. Watt et al. Vol. 8543. Lecture Notes in Computer Science. Springer, pp. 282–297.

— (2015). "SEPIA: Search for Proofs Using Inferred Automata". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, pp. 246–255.

Gransden, Thomas et al. (2016). "Revisit, Reuse, Recycle your Coq Proofs: Towards an Intelligent Interactive Proof Environment". In: *Journal of Automated Reasoning*. In preparation.

Grov, Gudmund, Ekaterina Komendantskaya, and Alan Bundy (2012). "A Statistical Relational Learning Challenge - extracting proof strategies from exemplar proofs". In: *ICML'12 worshop on Statistical Relational Learning*.

Harrison, John (1996). "Proof Style". In: *Types for Proofs and Programs: International Workshop TYPES'96*. Ed. by Eduardo Giménex and Christine Paulin-Mohring. Vol. 1512. Lecture Notes in Computer Science. Springer, pp. 154–172.

— (2006). "Floating-Point Verification Using Theorem Proving". English. In: *Formal Methods for Hardware Verification*. Ed. by Marco Bernardo and Alessandro Cimatti. Vol. 3965. Lecture Notes in Computer Science. Springer, pp. 211–242.

— (2009). "HOL Light: An Overview". In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Munich, Germany: Springer-Verlag, pp. 60–66.

Hastie, Trevor J., Robert John Tibshirani, and Jerome H. Friedman (2009). *The elements of statistical learning : data mining, inference, and prediction*. Springer series in statistics. New York: Springer.

Heras, Jónathan and Ekaterina Komendantskaya (2013). "ML4PG in Computer Algebra Verification". In: *Intelligent Computer Mathematics: MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*. Ed. by Jacques Carette et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 354–358.

Heras, Jónathan and Ekaterina Komendantskaya (2014). "Recycling Proof Patterns in Coq: Case Studies". In: *Mathematics in Computer Science* 8.1, pp. 99–116.

Jamnik, Mateja et al. (2003). "Automatic Learning of Proof Methods in Proof Planning". In: *Logic Journal of IGPL* 11.6, pp. 647–673.

Kaliszyk, Cezary, Lionel Mamane, and Josef Urban (2014). "Machine Learning of Coq Proof Guidance: First Experiments". In: *SCSS 2014 - 6th International Symposium on Symbolic Computation in Software Science*. Ed. by Temur Kutsia and Andrei Voronkov. Vol. 30. EPiC. EasyChair, pp. 27–34.

Kaliszyk, Cezary and Josef Urban (2013). "MizAR 40 for Mizar 40". In: *CoRR* abs/1310.2805. URL: `http://arxiv.org/abs/1310.2805`.

— (2014). "Learning-Assisted Automated Reasoning with Flyspeck". In: *Journal of Automated Reasoning* 53.2, pp. 173–213.

— (2015). "Learning-assisted theorem proving with millions of lemmas". In: *Journal of Symbolic Computation* 69. Symbolic Computation in Software Science, pp. 109 –128.

Klein, Gerwin et al. (2014). "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Transactions on Computer Systems* 32.1, 2:1–2:70.

Kohavi, Ron (1995). "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection". In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., pp. 1137–1143.

Komendantskaya, Ekaterina, Jónathan Heras, and Gudmund Grov (2013). "Machine Learning in Proof General: Interfacing Interfaces". In: *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012*. Ed. by Cezary Kaliszyk and Christoph Lüth. Vol. 118. EPTCS, pp. 15–41.

Kühlwein, Daniel and Josef Urban (2015). "MaLeS: A Framework for Automatic Tuning of Automated Theorem Provers". In: *Journal of Automated Reasoning* 55.2, pp. 91–116.

Lang, Kevin J., Barak A. Pearlmutter, and Rodney A. Price (1998). "Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm". In: *Proceedings of the 4th International Colloquium on Grammatical Inference*. ICGI '98. London, UK, UK: Springer-Verlag, pp. 1–12.

Leroy, Xavier (2009). "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7, pp. 107–115. ISSN: 0001-0782.

Matichuk, Daniel, Makarius Wenzel, and Toby Murray (2014). "An Isabelle Proof Method Language". In: *Interactive Theorem Proving: 5th International Conference,*

ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, pp. 390–405.

Meng, Jia and Lawrence C. Paulson (2009). "Lightweight relevance filtering for machine-generated resolution problems". In: *Journal of Applied Logic* 7.1. Special Issue: Empirically Successful Computerized Reasoning, pp. 41 –57.

Meng, Jia, Claire Quigley, and Lawrence C. Paulson (2006). "Automation for interactive proof: First prototype". In: *Inf. Comput.* 204.10, pp. 1575–1596.

Mitchell, Thomas M. (1997). *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc.

Naumowicz, Adam and Artur Korniłowicz (2009). "A Brief Overview of Mizar". In: *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 67–72.

Naumowicz, Adam and Artur Kornilowicz (2009). "A Brief Overview of Mizar". English. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 67–72.

Nipkow, Tobias, Markus Wenzel, and Lawrence C. Paulson (2002). *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag. ISBN: 3-540-43376-7.

Obua, Steven et al. (2014). "ProofPeer: Collaborative Theorem Proving". In: *CoRR* abs/1404.6186. URL: `http://arxiv.org/abs/1404.6186`.

Pearl, Judea (1984). *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley.

Quinlan, J. Ross (1993). *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Ron, Dana, Yoram Singer, and Naftali Tishby (1996). "The power of amnesia: Learning probabilistic automata with variable memory length". In: *Machine Learning* 25.2, pp. 117–149.

Russell, Stuart J. and Peter Norvig (2003). *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearson Education.

— (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson Education.

Schulz, Stephan (2013). "System Description: E 1.8". In: *Proc. of the 19th LPAR, Stellenbosch*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. LNCS. Springer.

Urban, Josef (2005). "MPTP – Motivation, Implementation, First Experiments". In: *Journal of Automated Reasoning* 33.3, pp. 319–339.

— (2013). "BliStr : The Blind Strategymaker". In: *CoRR* abs/1301.2683. URL: `http://arxiv.org/abs/1301.2683`.

Walkinshaw, Neil, John Derrick, and Qiang Guo (2009). "Iterative Refinement of Reverse-Engineered Models by Model-Based Testing". In: *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. Ed. by Ana Cavalcanti and Dennis R. Dams. Springer Berlin Heidelberg.

Walkinshaw, Neil, Ramsay Taylor, and John Derrick (2015). "Inferring extended finite state machine models from software executions". In: *Empirical Software Engineering*, pp. 1–43.

Walkinshaw, Neil et al. (2013). "STAMINA: a competition to encourage the development and assessment of software model inference techniques". In: *Empirical Software Engineering* 18.4, pp. 791–824.

Whiteside, Iain et al. (2011). "Towards Formal Proof Script Refactoring". In: *Intelligent Computer Mathematics*. Ed. by James H. Davenport et al. Vol. 6824. Lecture Notes in Computer Science. Springer, pp. 260–275.

Wieczorek, Wojciech (2017). *Grammatical Inference - Algorithms, Routines and Applications*. Vol. 673. Studies in Computational Intelligence. Springer.

Wiedijk, Freek (2008). "Formal proof – getting started". In: *Notices of the AMS* 55.11, pp. 1408–1414.

Witten, Ian H. and Eibe Frank (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd edition. San Francisco: Morgan Kaufmann.