UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 605

Regular Inference for Communication Protocol Entities

THERESE BOHLIN

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2009

ISSN 1651-6214
ISBN 978-91-554-7420-1
urn:nbn:se:uu:diva-9559

Dissertation presented at Uppsala University to be publicly examined in 2446, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Thursday, March 19, 2009 at 10:00 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Bohlin, T. 2009. Regular Inference for Communication Protocol Entities. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 605. 66 pp. Uppsala. ISBN 978-91-554-7420-1.

A way to create well-functioning computer systems is to automate error detection in the systems. Automated techniques for finding errors, such as testing and formal verification, requires a model of the system. The technique for constructing deterministic finite automata (DFA) models, without access to the source code, is called regular inference. The technique provides sequences of input, so called membership queries, to a system, observes the responses, and infers a model from the input and responses.

This thesis presents work to adapt regular inference to a certain kind of systems: communication protocol entities. Such entities interact by sending and receiving messages consisting of a message type and a number of parameters, each of which potentially can take on a large number of values. This may cause a model of a communication protocol entity inferred by regular inference, to be very large and take a long time to infer. Since regular inference creates a model from the observed behavior of a communication protocol entity, the model may be very different from a designer's model of the system's source code.

This thesis presents adaptations of regular inference to infer more compact models and use less membership queries. The first contribution is a survey over three algorithms for regular inference. We present their similarities and their differences in terms of the required number of membership queries. The second contribution is an investigation on how many membership queries a common regular inference algorithm, the L* algorithm by Angluin, requires for randomly generated DFAs and randomly generated DFAs with a structure common for communication protocol entities. In comparison, the DFAs with a structure common for communication protocol entities require more membership queries. The third contribution is an adaptation of regular inference to communication protocol entities which behavior foremost are affected by the message types. The adapted algorithm avoids asking membership queries containing messages with parameter values that results in already observed responses. The fourth contribution is an approach for regular inference of communication protocol entities which communicate with messages containing parameter values from very large ranges. The approach infers compact models, and uses parameter values taken from a small portion of their ranges in membership queries. The fifth contribution is an approach to infer compact models of communication protocol entities which have a similar partitioning of an entity's behavior into control states as in a designer's model of the protocol.

Keywords: Regular Inference, Automata Learning, Machine Learning, Communication Protocols, Parameterized Systems, State Machines, Deterministic Finite Automata, Mealy Machines

Therese Bohlin, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

© Therese Bohlin 2009

ISSN 1651-6214
ISBN 978-91-554-7420-1
urn:nbn:se:uu:diva-9559 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-9559)

To Magnus.

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **Model Checking**
by Therese Berg and Harald Raffelt. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, 2005, *Lecture Notes in Computer Science, Springer Verlag*, 3472:557-603. Revised version.

II **Insights to Angluin's Learning**
by Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. In Proceedings of the International Workshop on Software Verification and Validation, SVV 2003, *Electronic Notes in Theoretical Computer Science*, 118:3–18, 2005.

III **Regular Inference for State Machines with Parameters**
by Therese Berg, Bengt Jonsson, and Harald Raffelt. In Proceedings of the Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, *Lecture Notes in Computer Science, Springer Verlag*, 3922:107–121. Revised version.

IV **Regular Inference for State Machines using Domains with Equality Tests**
by Therese Berg, Bengt Jonsson, and Harald Raffelt. In Proceedings of the Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, *Lecture Notes in Computer Science, Springer Verlag*, 4961:317–331. Revised version.

V **Regular Inference for Communication Protocol Entities**
by Therese Bohlin, Bengt Jonsson. (2008) Technical Report 2008-024, Department of Information Technology, Uppsala University, September 2008. Revised version.

Reprints were made with permission from the publishers.

Comments on my Participation

Paper I

I am the sole author of the second half of Chapter 19, Section 19.4-19.5, and the co-author of the summary, Section 19.6.

Paper II

I did half the work regarding the implementation, conducting the experiments, and analyzing the results. I am a co-author to Section 4 and 5 in the paper.

Paper III

The algorithm presented in Paper III, was worked out in discussion with a co-author. I am the principal author of Section 5, and co-author of Section 3 and 6. I did half the work regarding implementing the algorithm, conducting the experiments, and analyzing the results.

Paper IV

The ideas to Paper IV are primarily worked out in discussion with a co-author. I am co-author to all sections of the paper.

Paper V

The ideas to paper V are worked out in discussion with a colleague. I am the sole author to the paper. I did a large part of the implementation work. I conducted the experiments and analyzed the results.

Other Publications

- **On the Correspondence Between Conformance Testing and Regular Inference**
  by Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.

- **Learnlib: a Library for Automata Learning and Experimentation**
  by Harald Raffelt, Bernhard Steffen, and Therese Berg. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, New York, NY, USA, 2005. ACM Press.

Acknowledgements

Foremost, I want to thank my supervisor Bengt Jonsson for patiently guiding me throughout my graduate studies. It is an inspiration, and motivation to work with a supervisor with his kind of enthusiasm and carefulness. I also want to thank my second supervisor Joachim Parrow for the general guidance in my studies, especially in writing this thesis.

I thank Harald Raffelt, and Bernhard Steffen for welcoming me to visit them in Dortmund for six months. Specially, I thank Harald for an interesting and fruitful collaboration, and his family for showing me great kindness and inviting me join them to a lot of activities during my stay in Germany.

I want to thank my research colleagues Olga Grinchtein, Johan Blom, Mayank Saksena, Anders Hessel, and Paul Pettersson at Polacksbacken for our discussions on work related issues. I am grateful for all colleagues at Polacksbacken, who made my time there a nice experience.

Finally, I want to thank my family Maj-Britt, Bengt-Olof, and Anna-Karin Berg for their important support during my graduate studies. I am fortunate to have my best friend, and likewise beloved husband, Magnus Bohlin, always by my side to support and motivate me.

Summary in Swedish

Reguljär inferens av kommunikationsprotokollenheter

I vår vardag händer det ofta att vi stöter på program som vi måste kunna hantera. Du använder ett program när du t.ex. ringer på din mobiltelefon, bokar en flygbiljett genom ett bokningssystem, eller slår på din miniräknare. Program är till för att underlätta vår vardag. Tyvärr händer det att de slutar fungera eller inte fungerar som de är tänkta att göra. Orsaken till detta är ofta att programmeraren av systemet har skrivit ett felaktigt program. Ett fel kan t.ex. uppstå när du lyckas boka den sista lediga flygbiljetten på ett flyg via ett bokningssystem, men vid samma tidpunkt lyckas en annan person boka exakt samma biljett; bokningssystemet har därmed inte hanterat två bokningar korrekt. Naturligtvis ligger det både i ditt och programmerarens intresse att fel som dessa upptäcks och rättas till innan systemet sätts i bruk. Den här avhandlingen syftar till att underlätta arbetet med att hitta fel i program, genom att automatisera en del av arbetet.

Det finns olika tillvägagångssätt för att hitta fel i system. Ett enkelt sätt är att en programmerare söker igenom programkoden. Detta kan vara ett effektivt sätt om det är en liten mängd programkod. Dessvärre är det svårt att använda samma tillvägagångssätt när programmet är större. *Testning* är en alternativ metod. I alla tekniker som är designade för att hitta fel i system, måste det åtminstone finnas någon idé om vad som är ett korrekt beteende hos ett system. Testning måste ha tillgång till en modell av hur systemet ska fungera, med andra ord en beskrivning av vad som är ett korrekt beteende hos systemet. Metoden jämför modellen med hur systemet verkligen fungerar genom att skapa en stor mängd så kallade testfall: input till systemet och hur systemet förväntas svara. Om det svarar som förväntat, så är vi nöjda, annars har vi hittat ett fel i systemet som måste åtgärdas.

Att automatisera testningen av ett system innebär att vi måste automatiskt kunna generera en modell och testfall. Det finns tekniker för att göra testing automatiserad givet en modell. Tyvärr kan det ofta vara fallet att det inte finns en tillräckligt noggrann modell, eller någon modell överhuvudtaget. I dessa situationer kan en teknik kallad *reguljär inferens* användas för att automatiskt skapa modeller av system. Reguljär inferens skapar modeller utifrån sekvenser av input till system och observationer av hur de svarar. Storleken på modellen och den tid det tar att skapa den växer desto större system och fler input systemet kan ta emot. Detta beror på att tekniken måste observera fler sekvenser av input och svar hos systemet för att kunna skapa en korrekt modell.

Vi har anpassat tekniken reguljär inferens till att skapa modeller av kommunikationsprotokoll. Ett kommunikationsprotokoll består av regler för formatet och sändningen av data. Enheter som använder sig av kommunikationsprotokollet interagerar med varandra genom att skicka och ta emot meddelanden som innehåller data. Meddelanden skickas mellan enheterna via en gemensam kommunikationskanal. Ett exempel på kommunikationsprotokoll är *Transmission Control Protocol (TCP)* (översatt till svenska: protokoll för överföringskontroll) som används för att skicka data mellan datorer. För vissa typer av protokoll, som till exempel TCP, kan vi tolka ett meddelande som att bestå av en meddelandetyp och ett antal parameterar. Till exempel, meddelandetypen *ACK* hos ett meddelande skickat ifrån enhet A till enhet B indikerar en bekräftelse på att A mottagit ett meddelande ifrån B. Andra delar i meddelandet kan vi tolka som parametrar, till exempel destinations-port och sekvensnummer. En modell av en kommunikationsprotokollenhet kan på grund av det stora antalet värden parametrar kan anta bli väldigt stor och därmed ta lång tid att skapa. Eftersom reguljär inferens inte utgår ifrån kommunikationsprotokollets programkod, utan istället ifrån dess beteende, så kan den skapade modellen bli väldigt olik programkoden.

I den här avhandlingen presenterar vi anpassningar av tekniken reguljär inferens som syftar till att

- skapa mer kompakta modeller av kommunikationsprotokollenheter som liknar programkodens struktur, och
- kräver att färre sekvenser av input och svar observeras.

Vi har undersökt och presenterat resultat kring en optimiering för kommunikationsprotokoll och likande system som syftar till att reducera antalet sekvenser av input och svar som måste observeras för att reguljär inferens ska kunna bygga modeller. Våra resultat visar att användandet av optimeringen på exempel av små kommunikationsprotokoll reducerar antalet observerade sekvenser av input och svar med kring 60%.

Vidare har vi anpassat reguljär inferens till kommunikationsprotokoll vars beteende till största del beror på meddelandetypen i meddelanden. Med denna anpassning krävs det färre observerade sekvenser av input och svar för att tekniken ska kunna bygga en modell.

Vissa parametrar är av typen identifierare, till exempel en adress till en resurs, och kan därför anta väldigt många olika värden. Vi har gjort anpassningar av reguljär inferens så att den kräver färre antal värden på denna typ av parametrar används i input. Trots det konstrueras fullständiga modeller, det vill säga alla värden för parametrarna finns i modellerna. De konstruerade modellerna är också kompakta.

Ofta är programkoden för kommunikationsprotokoll uppdelad i kontrolltillstånd, det vill säga programkod som har funktionalitet som programmeraren av programkoden tycker är konceptuellt lika. Vi har gjort en anpassning av reguljär inferens så att tekniken bygger modeller som har sitt beteende uppdelat i en struktur vilket liknar kontrolltillstånden hos programkoden som utgör kommunikationsprotokollet.

Contents

1. Introduction

1.1 Errors in Programs

In our daily lives we encounter programs which we are forced to handle. You use a program when you use your mobile telephone to make a call, book a flight ticket in a booking system, or use a pocket calculator. Programs facilitate our daily life. Unfortunately, it may happen that a program crashes or makes mistakes. The cause of this is often that the programmer of the system has written an incorrect program. An error can for instance occur when you succeed to book the last available flight ticket via a booking system, but at the same time someone else succeed to book the exact same ticket; the booking system has not handled two concurrent bookings correctly. Of course, both you and the programmer of the booking system want to discover these errors and correct them before the system is deployed. This thesis aims to facilitate the work of finding errors in programs, by automating some part of the work.

The type of systems that are considered in this thesis are so called *reactive systems*, systems that are usually intended to continuously receive input. Examples of such systems are web servers, communication protocols, operating systems, and controllers in embedded systems.

There are different means with which we can find errors in reactive systems. To manually inspect the program code of the system in order to find errors can be an efficient method in the cases the program is small. It is more difficult and takes more time to find errors when a program is extensive. *Testing* and *formal verification* are two alternative methods for finding errors. Both methods assume access to a so called *specification* of the system, i.e., a description of the correct behavior of the system. The methods compare the specification to the actual behavior of the system.

In testing, a so called *test case* is created, which consist of input to the system and the expected response (or output) from the system, according to the specification. The system is fed with the input in the test case; the response of the system is observed and then compared to the expected response. If the system and the test case have the same response, we say that the system has *passed* the test case, otherwise it has *failed*.

The method of formal verification often requires access to a formal *model*, which describes the behavior of the system. The model describes the system on a more abstract level, removing non-relevant details of the system. The model is used to verify that the system conforms with the specification. There are two well-established techniques for formal verification: *model checking* and *theorem proving*.

There is an important difference between formal verification and testing. In formal verification we can with certainty establish that that the formal model of the system conforms to the behavior that is described in the formal specification. In testing we can only execute a finite number of test cases, thus we can not completely exclude the possibility that there is an error in the system.

1.2 Specifications and Models of Systems

In all techniques designed to find errors in systems, there must be some idea of what a correct behavior of the system is. This applies to both testing and formal verification; they require access to a description of the intended behavior of the system, i.e., a specification. The specification can be formulated in a variety of formats. It can be an expert that decides whether or not the system has a correct behavior, a text document that describes a correct behavior of the system, a number of test cases, or a formal specification. A specification expresses how the system should work, and a model describes how the system actually works. The specification and the model can be expressed in the same form.

There are different types of formal specifications of reactive systems. Each type is an attempt to capture the characteristics of the system to test, or formally verify. Two common types of formal specifications are mathematical logic and state machines. Mathematical logic can be used to express models by means of inference from rules, e.g. modal logic and first-order logic. Other well-established techniques for formal specifications are the Specification and Description Language (SDL), a standardized language for specification and description of systems, specially with telecommunication systems in mind [BHS91], Estelle [ISO89a], an ISO standard similar to SDL, and LOTOS [BB89, ISO89b], an ISO standard for specifications of distributed systems based on the Calculus of Communicating Systems first presented by Milner [Mil80] in 1980. Other formal specification techniques for distributed systems are Communicating Sequential Processes (CSP) introduced by Hoare [Hoa78] in 1978, and Petri Nets, first introduced by Petri in the 1960s [Pet62]. A decade after CSP was introduced, Harel introduced statecharts which enables construction of specifications in hierarchical diagrams [Har87].

In this thesis we use several types of *state machines* for formal specifications and models of systems. A state machine contains states and transitions. It can be viewed as a graph in which there are nodes representing states, and labelled directed edges representing transitions. The state machine is at any time in a state, and can make a transition to a next state, via an outgoing edge labelled with a symbol. An *initial* state is indicated by an arrow with no origin pointing to the state, and represents where a computation of the state machine may start.

The simplest state machine model used in this thesis is called *deterministic finite automaton* (DFA). It just accepts or rejects sequences of symbols. Symbols can for example represent input and output of a system. A state in the DFA is either accepting or rejecting. If a sequence of symbols leads to an accepting state, the sequence is said to be *accepted* by the DFA, otherwise *rejected*. A DFA induce a so called *regular language*, which is the set of sequences of symbols that the DFA accepts. Figure 1.1 shows an example of a DFA with only accepting states, and where “Timer=Pp” is the initial state. It illustrates a simple model of a new communication protocol for the Korean railway signaling system [LJL$^+$07]. In state “idle” on input “operate_polling_timer” the DFA makes a transition to state “Timer=Pp”.

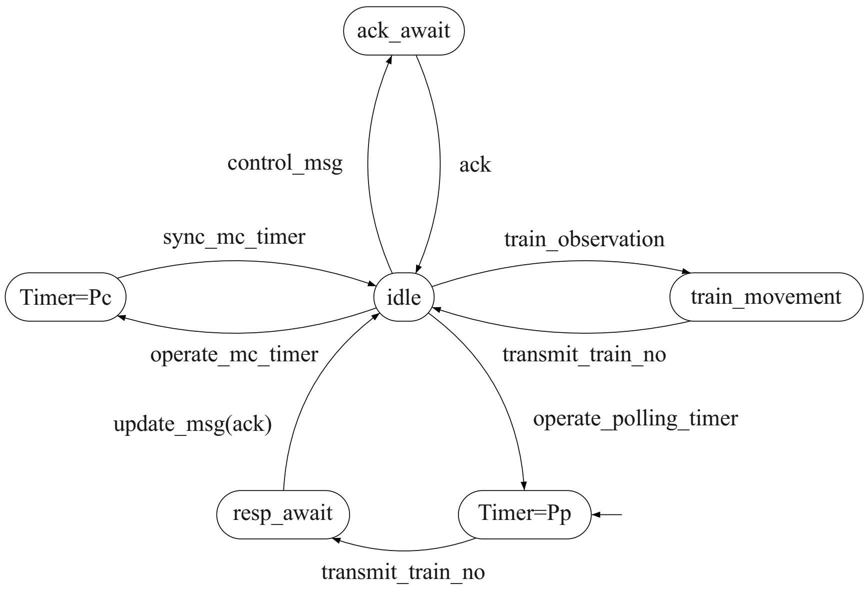

Figure 1.1: A Deterministic Finite Automaton model of a communication protocol.

A state machine model called *Mealy machine* suits systems which respond with output. It differs from the DFA model in that its transitions are labelled with both an input symbol and the corresponding output symbol produced by the system, instead of just a single symbol. E.g., a Mealy machine, shown in Figure 1.2, will in state q_0 on input “InitPollingReceived” output “ErrorCheck”, and put itself in state q_1.

A so called *Extended Finite State Machine* (EFSM) may have extensions to its states and transitions. Its states may for instance be extended with

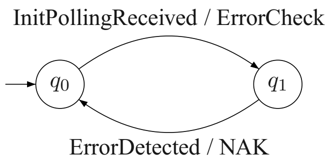

Figure 1.2: An example of a Mealy machine.

variables local to a state, in which values may be stored. Its transitions may for instance be labelled with assignments to variables, and guards consisting of boolean expressions which determine whether a transition can be made or not. The symbols in an EFSM may consist of several parts, for instance a parameterized symbol may consist of an action type together with a number of parameters, each of which can assume many values. A parameter can be symbolic, meaning that it represents several values. An example of an EFSM with two states q_0, and q_1, is shown in Figure 1.3. It has a location variable “PhoneNr” in state q_1. Its transitions are labelled with expressions consisting of parameterized input, guard, assignment to location variables, and parameterized output. The parameters are symbolic. The transition from state q_0 to state q_1 is labelled with the parameterized input “PhoneCall(phone_number, message)“, the guard “true”, the assignment of location variable “PhoneNr” to “phone_number”, and the parameterized output “ReceivedMessage(message)“. In state q_1 on parameterized input “PhoneCall(phone_number, message)“, a guard allows the transition back to state q_1 to be made if the value in “phone_number” is equal to the stored value in state variable “PhoneNr”. Otherwise the transition to state q_0 is enabled.

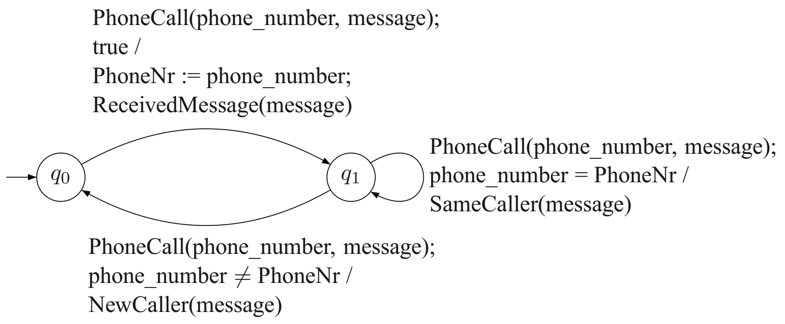
PhoneCall(phone_number, message);
true /
PhoneNr := phone_number;
ReceivedMessage(message)
q_0
q_1
PhoneCall(phone_number, message);
phone_number = PhoneNr /
SameCaller(message)
PhoneCall(phone_number, message);
phone_number $\neq$ PhoneNr /
NewCaller(message)

Figure 1.3: Example of an extended finite state machine.

1.3 Automated Error Detection

This section describes how theorem proving, model checking, and testing works in an automated fashion.

To begin with we need specifications and models in formats that can easily be made understandable to computers. Test cases and formal specifications belong to these types.

Theorem proving requires that both the model and the specification are expressed as formulas in some mathematical logic. The logic is given by a formal system, which defines a set of axioms and inference rules. Theorem proving is the process of inferring a proof of a property from the axioms of the system by applying the inference rules. There are interactive theorem provers that, assisted by a proficient human, prove that a model satisfies a specification. However, since they require human interaction they are slow.

Model checking typically assumes that the system is modelled by a state machine with propositions associated with each state. A proposition for a state represents an invariant that is valid when residing in that state. Model checking also requires access to a specification of the system, usually expressed in temporal logic. Temporal logic is a modal logic which is used to express formulas of propositions and time. The technique can automatically calculate a decision whether the model violates any property in the specification. An advantage with this technique, is that if a property does not hold a so called *counterexample*, a sequence of inputs to the state machine that makes it violate the property, is returned. The counterexample can be utilized by a user to discover errors in the model. A well-known problem with model checking is that the technique is computationally expensive, making it not applicable to large systems.

Completely automated testing involves automating the generation of a specification, test cases, and execution of test cases. Since automating test case execution is a system specific matter, let us focus on the more general aspects of test automation. Assuming we have a specification, testing can be made automated with a technique called *model-based testing*. From the specification a set of test cases, called *test suite*, can be generated.

Even though theorem proving, model checking, and testing may be fully automatic, often the task of constructing a specification and a model is manual. Since our goal is to automate error detection as much as possible, we opt for automatically constructing them.

There are different approaches for automatically creating specifications or models. In this thesis we focus on the construction of a model. By applying techniques like static analysis of a system's source code, a model can be created. Bandera [CDH$^+$00] is an example of an approach which combines different techniques of static analysis to automatically create a state machine model, and C2BP [BMMR01] is a tool to automatically create a mathematical model for C programs. But there are situations where there is no access to source code, e.g., third-party modules or libraries. However, so called *machine learning* techniques can be used to infer models by observing running systems.

1.4 Machine Learning

Machine learning is a group of algorithmic techniques that automatically constructs and refines models. The techniques are provided with a large set of data from which they construct models that classifies the data correctly and predict classifications of non-disclosed data. Particularly interesting for us are techniques to construct DFAs from sets of strings which are classified as accepted or rejected. Other examples of techniques are decision-tree induction and artificial neural networks. Decision-tree induction has successfully been used to predict pellet quality, and to determine whether to grant applications of credit cards or not [LS95]. Artificial Neural Networks is used for pattern recognition in a medical application to help cytotechnologists spot cancerous cells, for financial analysis in financial forecasting, and in control systems to detect spillage of molten steel before it occurs [WRL94].

Machine learning algorithms which infer models that accept regular languages are called *regular inference*. There are a number of algorithms which construct DFAs from samples of input sequences and the corresponding responses of the system [Ang87, BDG97, Dup96, Gol67, KV94, RS93, TB73]. They all infer a smallest DFA in the number of states in line with a philosophy called *Occam's razor*, which states that the smallest model that fits the collected samples is to prefer.

The regular inference technique can be used to construct DFA models of systems under test (SUT), by viewing sequences of input to the system as strings. Strings that cause the SUT to crash are interpreted as strings that should not be in the language accepted by a DFA model of the SUT. The regular inference algorithms infer a DFA from the answers to a finite set of *membership queries*, each of which asks whether a certain sequence of symbols is in the language accepted by the SUT or not. The algorithms use essentially the same basic principles. Given "enough" membership queries, the constructed automaton will be a correct model of the SUT. Angluin [Ang87] and others introduce *equivalence queries* which check whether the regular inference procedure is completed; if not, they are answered by a counterexample on which the current hypothesis and the SUT disagree.

In the 1960s Gold showed that it is possible to infer the regular language of a SUT with a finite number of wrong conjectures under certain circumstances [Gol67]. Since then several algorithms have been presented based on this result. Researcher have found vast application areas in which regular inference is useful. Alur et al. [AEY03] use regular inference to support designers of concurrent systems when constructing specifications in terms of Message Sequence Charts. Other application areas are to infer specifications [ABL02], infer assumptions on an environment of a component so that a certain property holds [CGP03], and to enable model-checking without a model of the SUT [GPY02]. We have focused our work on utilizing regular inference to automatically infer state machine models of communication protocol entities.

1.5 Communication Protocols

A communication protocol defines rules for the format and transmission of data. Entities of communication protocols interact by sending and receiving messages containing data. The messages are passed between entities over some common communication channel. Examples of communication protocols are the Internet Protocol (IP) and the Transmission Control Protocol (TCP). Typically, messages used by protocols in telecommunication applications consist of a Protocol Data Unit (PDU) type and a number of parameters. For example the TCP segment in an IP packet consist of 11 fields in the header, of which eight are flags, aka control bits, e.g., SYN, RST, and FIN. The control bits can be interpreted as PDU types which steer the control flow of a TCP entity. The other fields in the TCP header are for instance source port, sequence number, and acknowledgement number. Even though the control bits steer the control flow, parameters, such as acknowledgement numbers, also influence the control flow.

It is common that designers of communication protocols partition the functionality of a protocol into control states with state variables. In the state variables, values of messages parameters can be stored to be used to influence the behavior of the protocol, or used as parameter values in output messages.

The functionality in a communication protocol entity is often modelled by an EFSM. The EFSM shown in Figure 1.3 models the functionality of a communication protocol entity receiving phone calls. In the example an input symbol consists of the PDU type "PhoneCall", and values of the parameters "phone_number" and "message". The parameters are symbolic, which means that they may model several values. The EFSM also has a state variable called "PhoneNr" in state q_1. It is set on the transition from state q_0 to state q_1, and used in the guards labelling the looping transition in state q_1 and the transition from state q_1 to state q_0. All input symbols for which the value of the input parameter "phone_number" is the same as the number stored in the state variable "PhoneNr" will loop in state q_1 and produce an output symbol with PDU type "SameCaller", and all other input symbols will make a transition to state q_0 and output a symbol with PDU type "NewCaller".

1.6 Research Problems Addressed in This Thesis

In this section we present the research problems addressed in this thesis.

> *The focus of this thesis is on using regular inference techniques to infer state machine models of communication protocol entities.*

Communication protocols can be modelled with different types of state machines as described in Section 1.2; simple state machine models are DFAs and Mealy machines, and more advanced are EFSMs.

We intend to use regular inference to infer models of communication protocol entities. However, applying regular inference to communication protocols induce a number of problems. The number of symbols in state machine models of communication protocol entities are typically very large, since input messages contain parameters that range over possibly very large domains. The number of message sequences that have to be input to a communication protocol entity, i.e., membership queries, by a regular inference algorithm in order to infer a model, is therefore very large and takes a lot of time. A second issue is that the large number of input messages also may cause the simple state machine models of communication protocol entities to be very large.

> *The two problems we address in this thesis are that*
> - *regular inference techniques require a large amount of membership queries when inferring models of communication protocol entities, and*
> - *the inferred (simple) models of communication protocol entities are large.*

The large quantity of membership queries required by Angluin's L^* algorithm, was pointed out in the 1980s by Rivest and Schapire, as a property of L^* that needs to be addressed to make the technique practical for larger systems [RS89]. They imagine to use their algorithm in a real robot on a mission to learn its environment. They report that an experiment on a system with 400 states and 8 symbols required 130.000 membership and equivalence queries all together. Hungar et al. also report that the number of membership queries is a bottle-neck for L^*; a single membership query took them about 1.5 minutes when inferring a call center system, because of time-outs in the system [HNS03]. The largest system they inferred required about 132.000 membership queries, which they report would take them 4.5 months to execute. Therefore, they have suggested different types of domain-specific optimizations for L^* to reduce the number of membership queries required to infer a model [HNS03]. A particularly interesting domain-specific optimization is the one for DFA models that accept so called *prefix-closed languages*, which can be used to model reactive systems in general. A prefix-closed language contains all prefixes of a string in the language. There is an upper and lower bound on the number of membership queries required by L^*. It is of interest to find out where in span between the boundaries we can expect the L^* applied to a SUT. In the worst case the L^* algorithm requires $|\Sigma|mn^2$ number of membership queries, where $|\Sigma|$ is the number of symbols, m is the longest counter-example received in reply to an equivalence query, and n is the number of states in the model. In the best case L^* requires $|\Sigma|mn \log n$ number of membership queries. We have investigated if the average amount of membership queries required by L^* on DFAs accepting prefix-closed languages and general DFAs, is closer to the theoretical worst-case or best case of L^*. We have also investigated how much less membership queries L^* requires with the optimization for DFAs accepting prefix-closed languages.

As mentioned in Section 1.5, a basic property of communication protocols is that each of their input and output messages consists of a PDU type together with a number of parameters. In contrast, common regular inference algorithms have a rigid view of the input and output messages, viewing each message as a single unit: a symbol. The number of membership queries asked by these algorithms grows linearly in the number of input symbols, and for communication protocol entities the number of input symbols is exponential in the number of parameters in input messages. This makes the required number of membership queries grow exponentially in the number of parameters as well, and asking a large amount of membership queries takes time. Moreover, viewing input messages as single symbols also makes it difficult to interpret existing correlations between parameters and behaviors in the model. We have adapted the L^* algorithm to ask fewer membership queries whenever few parameters in input messages affect the behavior of the SUT.

Other difficulties, in inference of communication protocol entities, are induced by occurrences of parameters in input which take on values from very large domains. These type of parameters are for instance identity numbers, counters, and time stamps. However, the behavior of a communication protocol entity may not depend on the values of these type of parameters. A communication protocol is often data-independent in the sense that the parameters may only affect the entity's behavior depending on whether pairs of parameters have the same value. E.g., a communication protocol entity may behave differently, depending on whether a source address provided as a parameter in an input message is the same as the source address received in a preceding message. A simple state machine model of this type of communication protocol entity may be very large if the domain of at least one parameter in input is very large. We have constructed an approach to infer compact symbolic models of these type of data-independent communication protocol entities.

A second common property of a communication protocol, is that the protocol designer structures the model of the protocol into control states containing state variables in which input parameter values can be stored. It is appropriate to model this structure in an EFSM, since it is easy to incorporate control states and state variables in an EFSM. However, this structure of the protocol model is not taken into consideration by the common regular inference techniques. They infer flat simple state machine models with states that have the control state and the values of the state variables encoded in them. This makes the flat models not practical since they are large, and hard to correlate to the actual structure of the protocol model. Traditional methods do not infer EFSM models with control states and state variables, since these are not externally observable. If we would, in spite of this difficulty, infer an EFSM model in a naive way with state variables and control states, they may be very different from those of the protocol model. Assuming a model of a communication protocol entity is intended to be used to generate test suites based on some coverage criteria of the model, or be refined by a human in regression testing [HHNS02], a model with very different control states than those of the protocol model, is insufficient. We have constructed an approach for inferring EFSM models of communication protocol entities, such that the models are similar to the designer models of the entities.

1.7 Thesis Organization

The thesis is organized as follows. Chapter 2 gives a presentation of regular inference and work closely related to regular inference. Chapter 3 gives a summary of each paper included in this thesis together with a small discussion. Chapter 4 surveys related work. The last chapter, Chapter 5, presents the conclusions made in this thesis and points out interesting topics for future work. Thereafter follow reprints of the Papers I-V.

2. Regular Inference

In regular inference, we assume that we do not have access to the source code of the system we wish to model. In order to investigate the functionality of the system we observe the responses of the system to selected sequences of inputs. In this chapter we describe the regular inference technique. Let us first describe the established L^* regular inference algorithm for DFA by Dana Angluin [Ang87]. In Section 2.2 we present an adaption of the algorithm to inference of Mealy machines.

2.1 Regular Inference for DFA

In the setting of inferring DFA we assume that the response of the system is either that it executes on input or fails in some obvious way, for instance by crashing. We also assume the system to have a *reset*, which puts the system into its initial state. [1]

We assume a finite *alphabet* Σ of *symbols*. A *language* is a subset of Σ^*, the set of finite sequences of symbols, also called *strings*. A *deterministic finite automaton (DFA)* $\mathcal{M}$ over Σ is a tuple (Q, δ, q_0, F), where Q is a non-empty finite set of *states*, $\delta : Q \times \Sigma \to Q$ is the *transition function*, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *accepting states*. The transition function is extended from input symbols to strings of input symbols in the standard way, by defining $\delta(q, \varepsilon) = q$, and $\delta(q, ua) = \delta(\delta(q, u), a)$.

A string u is *accepted* iff $\delta(q_0, u) \in F$. The *language* accepted by $\mathcal{M}$, denoted by $\mathcal{L}(\mathcal{M})$, is the set of accepted strings. A subset $\mathcal{L} \subseteq \Sigma^*$ is said to be *regular* if $\mathcal{L}$ is accepted by some DFA. A language $\mathcal{L}$ is *prefix-closed* if for every w in $\mathcal{L}$, all prefixes of w are in $\mathcal{L}$. We say that a DFA is prefix-closed if its language is prefix-closed. A minimal prefix-closed DFA has exactly one non-accepting state.

We here give a succinct description of the main ideas behind regular inference. We assume that a system in which we are interested can be modeled by a DFA $\mathcal{M}$. The problem can now be looked upon as identifying the regular language which is accepted by $\mathcal{M}$, denoted by $\mathcal{L}(\mathcal{M})$.

[1]Assuming that the system is strongly connected, that is, there is a directed path between every pair of states in the system, a model can be generated without the need of reset [RS93].

In a learning algorithm a so called *Learner*, who initially knows nothing about $\mathcal{M}$, is trying to learn $\mathcal{L}(\mathcal{M})$ by asking queries to a *Teacher* and an *Oracle*. There are two kinds of queries.

- A *membership query* consists in asking the *Teacher* whether a string $w \in \Sigma^*$ is in $\mathcal{L}(\mathcal{M})$.
- An *equivalence query* consists in asking the *Oracle* whether a hypothesized DFA $\mathcal{A}$ is correct, i.e., whether $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{M})$. The *Oracle* will answer yes if $\mathcal{A}$ is correct, or else supply a counterexample u, either in $\mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{A})$ or in $\mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{M})$.

The typical behavior of a *Learner* is to start by asking a sequence of membership queries, and gradually build a hypothesized DFA $\mathcal{A}$ using the obtained answers. When the *Learner* feels that she has built a stable hypothesis $\mathcal{A}$, she makes an equivalence query to find out whether $\mathcal{A}$ is correct. If the result is successful, the *Learner* has succeeded, otherwise she uses the returned counterexample to revise $\mathcal{A}$ and perform subsequent membership queries until arriving at a new hypothesized DFA, etc. We discuss the realization of the *Oracle* in Section 2.6.

2.1.1 The L^* Algorithm

The information accumulated by the L^* algorithm is a finite collection of observations, which is organized into an observation table. An *Observation Table* over a given alphabet Σ is a tuple $\mathcal{OT} = (S, E, T)$, where

- $S \subseteq \Sigma^*$ is a nonempty finite prefix-closed set,
- $E \subseteq \Sigma^*$ is a nonempty finite suffix-closed set, and
- $T : ((S \cup S \cdot \Sigma) \times E) \to \{+, -\}$ is a (finite) function satisfying the property that $se = s'e'$ implies $T(s, e) = T(s', e')$ for $s, s' \in S \cup S \cdot \Sigma$ and for all $e, e' \in E$.

The strings in $S \cup S \cdot \Sigma$ are called *row labels* and the strings in E are called *column labels*. Each entry consists of a sign $+$ or $-$, representing whether a string is accepted or not.

The observation table is divided into an upper part indexed by S, and a lower part indexed by all strings of the form sa, where $s \in S$ and $a \in \Sigma$, that do not already appear in the upper part. Moreover the table is indexed column-wise by a suffix-closed set E of strings. The function T maps a row label s and a column label e, i.e. $T(s, e)$, to the set $\{+, -\}$, the algorithm will ensure that it is $+$ if $se \in \mathcal{L}(\mathcal{M})$ and $-$ otherwise.

For every $s \in (S \cup S \cdot \Sigma)$, a function $row(s)$ denotes the finite function from E to $\{+, -\}$, defined by $row(s)(e) = T(s, e)$. In other words, $row(s)$ is the row of entries in the observation table for row label s.

A distinct row of entries $row(s)$, where $s \in S$, characterizes a state in the DFA, which can be constructed from $\mathcal{OT}$. The rows of entries labeled by elements of $S \cdot \Sigma$ are used to create the transition function for the DFA.

To construct a DFA from the observation table it must fulfill two criteria. It has to be *closed* and *consistent*. An observation table $\mathcal{OT}$ is *closed* if for each $s \in S \cdot \Sigma$ there exists an $s' \in S$ such that $row(s) = row(s')$. An observation table is said to be *consistent* if whenever $row(s) = row(s')$ for $s, s' \in S$ then $row(sa) = row(s'a)$ for all $a \in \Sigma$.

When the observation table $\mathcal{OT}$ is closed and consistent it is possible to construct the corresponding DFA $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ as follows:

- $Q = \{row(s) \mid s \in S\}$, note: the set of *distinct* rows,
- $q_0 = row(\varepsilon)$,
- $F = \{row(s) \mid s \in S \text{ and } T(s, \varepsilon) = +\}$,
- $\delta(row(s), a) = row(sa)$.

The corresponding DFA constructed in this manner from table $\mathcal{OT}$ is denoted $\mathcal{A}(\mathcal{OT})$.

The L^* algorithm maintains the observation table $\mathcal{OT}$. The sets S and E are both initialized to $\{\varepsilon\}$. Next the the algorithm performs membership queries for ε and for each $a \in \Sigma$, the result is a sign for each queried string. The observation table $\mathcal{OT}$ is initialized to (S, E, T).

Next the algorithm makes sure that $\mathcal{OT}$ is closed and consistent. If $\mathcal{OT}$ is not consistent, one inconsistency is resolved through finding two strings $s, s' \in S$, $a \in \Sigma$ and $e \in E$ such that $row(s) = row(s')$ but $T(sa, e) \neq T(s'a, e)$, and adding the new suffix ae to E. The algorithm fills the missing entries in the new column by asking membership queries.

If $\mathcal{OT}$ is not closed the algorithm finds $s \in S$ and $a \in \Sigma$ such that $row(sa) \neq row(s')$ for all $s' \in S$, and adds sa to S. The missing entries in $\mathcal{OT}$ are inserted through membership queries.

When $\mathcal{OT}$ is closed and consistent the hypothesis $\mathcal{A} = \mathcal{A}(S, E, T)$ can be formed and its correctness checked through an equivalence query to the *Oracle*. The *Oracle* can either reply with a counterexample t, such that $t \in \mathcal{L}(\mathcal{M}) \iff t \notin \mathcal{L}(\mathcal{A})$, or 'yes'. If the answer is 'yes' the algorithm halts and outputs the correct conjecture $\mathcal{A}$. Otherwise t is a counterexample. Angluin's algorithm adds t and all its prefixes to S. Then it asks membership queries for the missing entries.

2.2 Regular Inference for Mealy Machines

Niese has presented an adaptation of Angluin's L^* algorithm for inference of Mealy machines [Nie03]. In general the setting for the adapted algorithm is assumed to be the same as for L^*. The algorithm has access to a membership and equivalence oracle, and collects the response from the SUT in an observation table. The algorithm also asks membership queries in the same manner as L^* does, and constructs conjectures whenever it can construct a stable model. The difference to the setting for L^* is that instead of observing whether the SUT accepts or rejects input, the adapted algorithm observes the output symbols the SUT produces in response to input.

A Mealy machine is a tuple $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ where I is a nonempty set of *input symbols*, O is a finite nonempty set of *output symbols*, Q is a nonempty set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times I \to Q$ is the *transition function*, and $\lambda : Q \times I \to O^*$ is the *output function*. Elements of I^* and O^* are (input and output, respectively) *strings*.

Now let us describe how Angluin's L^* algorithm is adapted by Niese to inference of Mealy machines. We assume that the SUT can be described by the unknown Mealy machine $\mathcal{M}_U = \langle I, O_U, Q_U, q_0^U, \delta_U, \lambda_U \rangle$. In the description of the inference algorithm for Mealy machines, we exchange all occurrences of the alphabet of symbols Σ to the alphabet of input symbols I. The set of suffixes E in the observation table is in this setting initialized to I. The response from the SUT is now sequences of output symbols from O_U. This is reflected in the entries of the observation table, which will contain strings of output symbols from O_U^* instead of $\{+, -\}$. We modify the function T so that $T : ((S \cup S \cdot \Sigma) \times E) \to O_U^*$ maps from row and column labels to strings of output symbols O_U^*, and define $T(s, ea)$ to be o if $\lambda_U(\delta_U(q_0^U, se), a) = o$, where $s \in S$, $ea \in E$, $a \in I$, and $o \in O_U^*$. We also modify the function $row(s)$, so that for each $s \in (S \cup S \cdot I)$ it denotes the finite function $row(s) : E \to O_U^*$ defined by $row(s)(e) = T(s, e)$.

Once the observation table $\mathcal{OT}$ is closed and consistent it is possible to construct a hypothesis $\mathcal{H} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ as follows:

- $O = \{T(s, a) \mid s \in S, a \in I\}$,
- $Q = \{row(s) \mid s \in S\}$,
- $q_0 = row(\varepsilon)$,
- $\delta(row(s), a) = row(sa)$, and
- $\lambda(row(s), a) = T(s, a)$.

The hypothesis $\mathcal{H}$ is provided in an equivalence query. The *Oracle* responds, as in the L^* algorithm, with a "yes" or a counterexample. However, a counterexample is this setting an input sequence $w \in I^*$, for which the SUT $\mathcal{M}_U$ and the hypothesis $\mathcal{H}$ produce different output $\lambda_U(q_0^U, w) \neq \lambda(q_0, w)$.

2.3 Other Regular Inference Algorithms for DFA

There exist a handful of regular inference algorithms. There are other algorithms that are rather similar to Angluin's L^* algorithm. One is an algorithm by Rivest and Schapire [RS93] that uses a *reduced observation table*. Compared to the observation table this reduced version stores a smaller portion of queries and answers. The requirement on the row indices is relaxed so that the set is not required to be prefix-closed. A third alternative by Kearns and Vazirani [KV94] uses a completely different data structure to store information, a binary *discrimination tree*. The nodes of the tree contains suffixes which are used as before to distinguishing prefixes that lead to different states from another. Balcázar et al. has presented a unifying concept from which these algorithms, including L^*, can be viewed [BDG97]. In essence, the algorithms differ in how many membership queries are required before a model is constructed. Among these three, the L^* algorithm generally performs the largest number of membership queries before a model is created. But because L^* collects more information before generating a model, it is also more likely to produce fewer false hypotheses and thus fewer equivalence queries. The upper bound on the number of equivalence queries is however the same for all three algorithms.

2.4 Complexity

The complexity of the algorithms is most often measured in the number of required membership and equivalence queries. The reason for this is that executing a membership query involves interaction with the system, and this is likely to require some time. The observation table, or other data-structure for queries and answers, also needs to be stored; for that we need to allocate memory resources. In the following, let n be the number of states in of the minimal DFA or Mealy machine model of the SUT, let m be the length of the longest counterexample returned in an equivalence query, let $|\Sigma|$ be the size of the alphabet of symbols Σ, and let $|I|$ be the size of the alphabet of input symbols I.

Let us first start with the number of equivalence queries. In L^*, Niese's adaptation of L^* to Mealy machines, and the algorithms using a reduced observation table or discrimination tree, the upper bound on the number of equivalence queries is n.

The upper bounds on the number of membership queries ($O(\mathit{Memb.Q.})$) are more diverse for these algorithms, they are shown in Table 2.1. The bounds for membership queries depend on whether answers to queries are saved, and how many membership queries are asked before creating a conjecture. Among the presented algorithms only the discrimination tree algorithm does not save the answers.

In practice the three inference algorithms would all perform poorly when applied to large systems according to upper bounds. We illustrate this with an example. In our earlier work [BJLS05] the model of an ATM protocol, shipped with the Edinburgh Concurrency Workbench [MS], used for experiments has 1715 states and 27 symbols. Assuming that the longest counter-example is 1715 symbols long, the effort of applying Angluin's algorithm to this particular example is asking $27 \times 1715^2 \times 1715 = 1.4 \times 10^{11}$ membership queries in the worst case. Thus, optimizations are necessary for the regular inference to work well in practice.

Algorithm	$O(\mathit{Memb.Q.})$
Angluin's L^*	$\|\Sigma\|n^2m$
Mealy Machine	$max(n, \|I\|)\|I\|nm$
Reduced Observation Table	$\|\Sigma\|n^2 + n\log m$
Discrimination Tree Table	$\|\Sigma\|n^3 + nm$

Table 2.1: *The upper bound on the number of membership queries for the regular inference algorithms, where n is the number of states in a minimal model of the SUT, m is the length of the longest counterexample, $|\Sigma|$ is the size of the alphabet Σ, and $|I|$ is the size of the input alphabet I.*

2.5 Optimizations

A suggestion for optimizations by Hungar et al. [HNS03] is based on the idea that knowledge about the domain of the system can be used to reduce the number of membership queries required by a regular inference algorithm. One optimization exploits that instances of communicating processes may behave in the same way and therefore can be interchanged. E.g., two telephones behave in the same way, therefore it is enough to investigate the behavior of one of them. A second employs knowledge about reactive systems, modeled as finite state machines with prefix-closed languages. They have evaluated their suggestions for optimizations on 7 examples. With these examples they have accomplished a total reduction in membership queries varying between 87% to 99.8% using all three optimizations. Applying only the optimizations for prefix-closed systems, they saved on average approximately 74% membership queries.

2.6 Equivalence Oracle

In the regular inference setting we require an equivalence oracle. The oracle's job is to either confirm that the suggested conjecture is correct or provide a counter-example. There is however no magical oracle that will provide this information for free. The oracle is a theoretical construction to make an idealization of a potentially hard problem, in order to provide a clean setup in regular inference. In practice, ways to provide counter-examples are for instance by monitoring the system and collecting a counter-example whenever the model and system disagrees, by letting a system expert evaluate the model, or by testing the system with randomly generated tests, or tests from a so called conformance test suite (see Section 2.7 for details regarding conformance testing). But all of these mimicked oracles have their disadvantages; monitoring may produce a very long counter-example which affects the complexity of the regular inference algorithm negatively, involving a system expert makes the regular inference technique only semi-automated and therefore less attractive, and finding a counter-example by executing randomly generated tests or tests from a conformance test suite is like executing membership queries. The advantage of conformance testing is that it provides a systematic way of achieving an answer to an equivalence query. Let k be the number of states in the DFA hypothesis of the system, and assume that we have an upper bound, l, on the number of states of a minimal DFA that models the system. Then if $l > k$, by applying the tests in a conformance test suite by Vasilevski and Chow [Cho78, Vas73] (VC) to the system we will find at least one test that the system does not pass. This test constitutes a counter-example as answer of an equivalence query. According to Vasilevski [Vas73], an upper bound for the total length of such a test sequences suite is $O(k^2 l |\Sigma|^{l-k+1})$, i.e., it is exponential in the difference between the number of states of the system and the hypothesis.

2.7 Regular Inference in Relation to Conformance Testing

In all model-based techniques, there is the problem of how to check that the model is an accurate description of the system. In the black box setting, a way to check that the model is equivalent to the system is by a technique called *conformance testing*. A conformance testing technique generates a set of tests, T, a so called *test suite*. A test, $t \in T$, consists of a string and the expected output of the system in response to the string. The system is said to *pass* a test if the actual output is the same as the expected output. We can confirm that a model is correct with respect to the system if the system passes all tests in a conformance test suite, and the system and model satisfies required hypotheses. Examples of required hypotheses are that the model is minimized, the system does not change during testing, the transition function for the model is total, and there is a known upper bound on the number of states in the system. If the system fails to pass a test t, then t is a counter-example to the conjecture that the model and system are equivalent.

A usual hypothesis is that the number of states of the system is exactly as many as the number of states of the model. Conformance test techniques that can be used in this setting are the W [Cho78, Vas73], Wp [FvBK$^+$91], and Z [LY96] techniques. The main difference between them is that the Wp and Z techniques generate a smaller test suite compared to the W technique. Test suites generated by these techniques are similar to the set of strings asked for in membership queries by regular inference algorithms. In our earlier work [BGJ$^+$05] we have performed a closer investigation of the relationship between conformance testing and regular inference. Assume that the model $\mathcal{A}$ has k states, and the system $\mathcal{M}$ has l states. We show in [BGJ$^+$05] that the set of strings in an observation table, UV, by which the L^* algorithm generates the DFA $\mathcal{A}$ is also a conformance test suite for $\mathcal{A}$, generated by the W technique. Consequently, under the assumption that $k = l$, there is actually no need for an equivalence test since we then know that we have a correct model; this also follows from [Ang87]. Assume that $\mathcal{M}$ has more states than $\mathcal{A}$, i.e. $l > k$, then using a conformance test suite as an equivalence oracle in a regular inference setting, we see the possibility to generate a conformance test suite based on the strings in UV. A conformance test suite based on the W technique is for $\mathcal{M}$ the set $U(\varepsilon \cup \Sigma \cup \Sigma^2 \cup \ldots \cup \Sigma^{l-k})V$ [Cho78]. If $\mathcal{A}$ and $\mathcal{M}$ disagree, at least one counter-example will be found among the set of tests $U(\varepsilon \cup \Sigma \cup \Sigma^2 \cup \ldots \cup \Sigma^{l-k})V$ with the exception of UV, so there is no need to test these again.

2.8 Regular Inference together with Model Checking

The *Model checking* [BJK$^+$04, Hol97] technique automatically checks that models of systems have a given property. A technique to model check a black box is presented by Peled et al. [PVY99] by combining regular inference and model checking. The idea of combining the two techniques is further elaborated to a method called *adaptive model checking* [GPY02].

In the setting for adaptive model checking we have access to the system, but seen as a black box, and a property we want to check on the system. The basic idea in adaptive model checking is to apply a regular inference algorithm to create a model of the system and then apply a model checking algorithm to check whether the model satisfies the property. An overview of how the algorithms are combined is shown in Figure 2.1. If the property holds, the regular inference algorithm makes an equivalence query with the conjecture. If the oracle replies with a counter-example, it is inserted into the regular inference algorithm and the model is revised.

In the case the property does not hold for the model, the model checker provides a counterexample, a string that when the model traverses it, the model violates the property. The adaptive model checking algorithm inputs the counterexample in the system and observes whether it can be executed by the system or not. If it can be, the algorithm concludes that the system does not satisfy the property, otherwise the counterexample is fed back to the regular inference algorithm as a counterexample to refine the model.

When the property holds and the oracle confirms that the model is correct, the adaptive model checking algorithm terminates and concludes that the system satisfies the property.

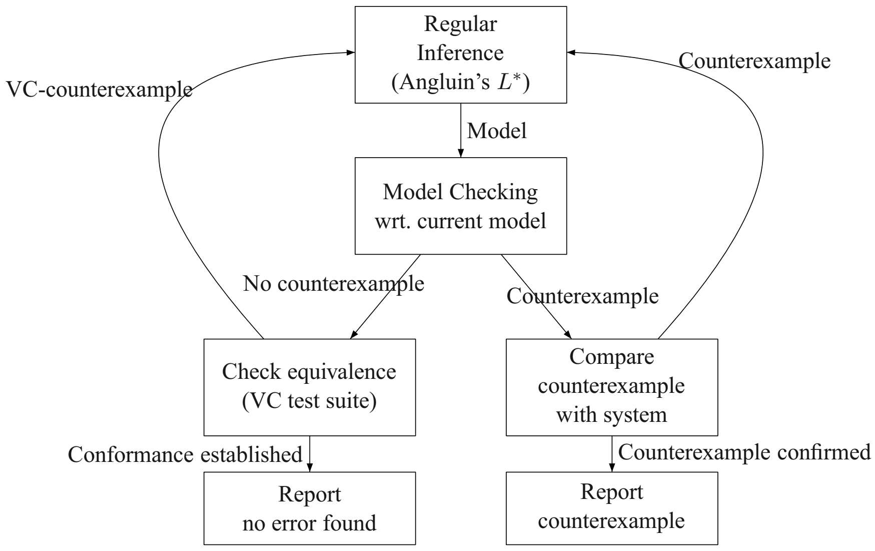

Figure 2.1: Overview of the Adaptive Model Checking Algorithm.

3. Summary of Papers

This chapter contains summaries of, and retrospective discussions about the work in the papers included in the thesis.

Paper I: Learning Finite State Machines

Paper I is included in a book on model-based testing of reactive systems [BR04]. The paper contains a survey of well-known techniques for regular inference. It is intended as a tutorial on the subject of regular inference. The survey contains presentations of Angluin's L^* algorithm [Ang87], the algorithm using a reduced observation table by Rivest and Schapire [RS93], and the algorithm using a discrimination tree by Kearns and Vazirani [KV94]. In the paper, these algorithms are presented using the unifying concept of observation packs, presented by Balcázar et al. [BDG97]. The presentation of the algorithms contains a theoretical background, their complexity, the algorithms presented in pseudo code, and illustrating examples. The survey also suggests two different ways to realize equivalence queries.

Furthermore, the paper contains an introduction to domain-specific optimizations in regular inference by Hungar, Niese, and Steffen [HNS03]. These optimizations are based on the idea that knowing certain properties of the system to be inferred, can be used to reduce the number of membership queries required by a regular inference algorithm. The adaptive model checking technique to combine regular inference and model checking, described in Section 2.8, is also presented in this paper.

Practical results collected from papers on applying Angluin's L^* algorithm on adaptive model-checking and domain-specific optimizations to examples [GPY02, HNS03, BJLS05] are presented in short.

Discussion

The main goal in this paper is to introduce the reader to common regular inference algorithms for inferring models of reactive systems. In order to also point out the large similarities between the algorithms we used the unifying concept of observation packs, presented by Balcázar et al. [BDG97]. We also describe the application of the algorithms to an example and put the algorithms into pseudo code.

In the paper upper bounds on the number of membership and equivalence queries for the algorithms are given, however results from experiments are not presented for all algorithms. We think experimental results from applying the algorithms to the same set of examples would have improved the presentation of the approaches. The reader would then have been able to compare the approaches to one another in terms of membership and equivalence queries, and space and time consumption. This would however required access to implementations that do not exist, and would require the time and effort of a separate paper.

Paper II: Insights to Angluin's Learning

We have implemented the L^* algorithm for regular inference proposed by Dana Angluin. The implementation is created in a straight-forward manner in Java and uses the library AMoRE [MMP$^+$95] for manipulating DFA. It is used to analyze the performance of the L^* algorithm on randomly generated examples, and a handful of protocols supplied with the Edinburgh Concurrency Workbench (CWB) [MS]. The focus of our analysis is foremost on the number of membership queries required by the algorithm on different types of examples. We measure complexity in terms of membership queries, since they are likely to be the most time-consuming activity when using regular inference in practice.

We have experimented with two types of randomly generated examples. The DFA in the first set of examples are randomly generated with respect to transitions and the percentage of accepting states. Let n, $|\Sigma|$, and m be as described in Section 2.4, and let $|\delta| = n|\Sigma|$ be the number of transitions. In our experiments we assume that equivalence queries always return a shortest possible counter-example, hence the length m of the longest counter-example is at most n. On our examples the number of membership queries grows approximately linearly with the number of transitions, $|\delta|$. This result is in contrast to the upper bound of the algorithm in this setting, which is $n^2|\delta|$, i.e. cubic in the number of states.

The second type of randomly generated examples is constructed in a way to imitate examples of reactive systems. All examples are prefix-closed DFAs. The number of membership queries for these type of examples grows approximately quadratically in the number of transitions, $|\delta|^2$. This is closer to the upper bound of the algorithm in comparison to the first type of examples.

We hoped to improve this result with the aid of a domain-specific optimization for prefix-closed systems. The optimization makes use of knowledge about prefix-closed languages: extensions of not accepted strings are not accepted. This resulted in an approximately 20% reduction of membership queries on randomly generated examples. Applying one of the two optimizations for prefix-closed systems, presented by Hungar et al. [HNS03], to the protocols provided in CWB, we attained a 59% reduction of the required number of membership queries. Hungar et al. [HNS03] present approximately the same level of reduction, 74%, of the number of membership queries using the two optimizations for prefix-closed systems.

We also compared Angluin's algorithm applied to a small set of examples supplied by CWB to the algorithm applied to a set of randomly generated prefix-closed DFA. We found that on average the CWB examples required 7% fewer membership queries without and 35% with the optimization, in comparison to the randomly generated prefix-closed DFA with the same number of states and symbols. We think the reason for this is that the CWB examples has relatively few transitions leading to accepting states, compared to the randomly generated examples.

Discussion

We are particularly interested to know how the L^* algorithm performs on systems in general, and particularly on reactive systems. Therefore we investigate how the L^* algorithm scales with size of systems; in particular, how it performs on prefix-closed DFAs, since prefix-closed DFAs can be used to model reactive systems. Our results for prefix-closed DFAs show that these generally require more membership queries than general DFAs with the same number of states and symbols. We believe that the cause for this difference in the number of required membership queries is that it is harder to distinguish states from each other in prefix-closed DFAs.

Our implementation is coded in Java and no particular effort was spent to optimize the data structures. Hence we see room for optimizations in the time and memory consumption. Optimizing the implementation would have allowed us to experiment with larger models than we did. On the other hand, the trend for the different categories of examples were clear and conclusions regarding the growth of membership queries could be drawn from the results.

Paper III: Regular Inference for State Machines with Parameters

In Paper II we experienced that models of reactive systems required a larger amount of membership queries compared to randomly generated examples with the same number of states and symbols. From this point on, our work shifted towards domain-specific optimizations. We have developed a domain-specific optimization intended to work more efficiently on a particular class of systems of interest, entities of communication protocols.

The technique of regular inference can be used to construct models of communication protocol entities. Typically in such protocols symbols consists of a PDU type together with a number of parameters, each of which can assume several values. Hence, the alphabet of such an entity is exponential in the number of parameters. The large alphabet makes the regular inference technique unattractive since it negatively affects the required amount of membership queries. If only a few of the parameters determine the behavior of the system in a given state, it is desirable to avoid asking membership queries for the parameter settings that have no affect. Therefore we have developed a regular inference algorithm that specializes in handling these cases [BJR06]. We have implemented this algorithm in the framework of LearnLib [RSB05], a library for automata and regular inference.

Our algorithm infers EFSM models. The states in the EFSM model are either accepting or rejecting, and each transition in the model is labelled with a PDU type together with a number of symbolic parameters, and a conjunction over negated and not negated symbolic parameters. The conjunction represents a guard for when the transition is enabled.

The idea behind our specialized algorithm is to infer, for each state and each PDU type, a partitioning of input symbols into equivalence classes, under the hypothesis that all input symbols in an equivalence class have the same affect on the state machine. Initially all input symbols of a PDU type are put in one equivalence class. Whenever such a hypothesis is disproved, equivalence classes are refined. In the EFSM that is inferred by our algorithm, the transitions are labelled with such equivalence classes. The number of membership queries the algorithm require is bounded by $O(nkm)$, where n is the number of states in the EFSM, k is the number of transitions of our EFSM, and m is the longest counter-example received in response to an equivalence query. The result of applying the algorithm to a large set of randomly generated examples shows that the algorithm requires less membership queries compared to L^* when the number of transitions in the EFSM is relatively small.

We also formed a second idea to further optimize Angluin's regular inference algorithm. The idea is to reduce the number of membership queries required to construct a model. We realize this by relaxing the requirement that every entry in the observation table must be filled, we present weaker restrictions, which preserve important properties of L^*.

The two optimizations form our regular inference algorithm for state machines with parameters. We implemented our algorithm in LearnLib [RSB05], in order to make a comparison between our and Angluin's L^* algorithm.

We focused on generating and running synthesized examples that varied the number of equivalence classes of input symbols. The result from exercising our algorithm on the examples, illustrates how the number of membership queries grows in proportion to the number of parameters affecting the state machine. This is the result we had hoped for; when there is a low proportion of input symbols affecting the system, we can reduce a large amount of membership queries by applying our algorithm. However, the negative result is that the required number of equivalence queries is larger with our algorithm. This can be explained by the fact that with our algorithm, models can be generated based on fewer membership queries, i.e., less information; hence a larger number of incorrect models may be created.

Discussion

The result presented in this paper is the outcome of the two optimizations combined into a new algorithm. The optimizations can theoretically be applied separately but they are not so in this investigation. It may be of interest to insert this separation of the optimizations into the implementation in order to see the contribution of each optimization.

We believe that the large number of equivalence queries required by our algorithm contributes to the growth of membership queries. The reason is that Angluin's algorithm requires that all prefixes of counter-examples must be queried for and also all one symbol extensions of the prefixes.

Paper IV: Regular Inference for State Machines using Domains with Equality Tests

In Paper IV we continue our work to optimize regular inference techniques for entities of communication protocols. As in Paper III we assume that such entities typically communicate by messages that consist of a PDU type with a number of parameters, each of which ranges over a sometimes large domain. In order to fully support the generation of models with data parameters, we have in Paper IV, worked out a general theory for inference of infinite-state state machines with input and output symbols from potentially infinite domains.

We present (to our knowledge) the first extension of regular inference to infinite-state state machines. In this work we consider Mealy machines where input and output symbols are constructed from a finite number of PDU types together with parameters from potentially infinite domains. We consider the type of systems where the only allowed operation on parameter values is a test for equality. The motivation is to handle parameters that, e.g., are identifiers of connections, objects, etc. The inferred model is an EFSM. The EFSM model has states called control locations, which may contain location variables. Each transition in the EFSM is labelled with four parts. Two parts consist each of a PDU type together with a number of symbolic parameters, representing input and output. The third part is a number of tests for equality between parameter values, representing a guard for when the transition is enabled. The last part is an assignment of parameter values from input to location variables. Parameters are stored in location variables of the EFSM for later use.

In standard regular inference states and transitions are inferred, and counterexamples to a hypothesized DFA are only used to add more states to the DFA. In Paper IV, we also infer control locations, location variables and operations on them, and counterexamples to a hypothesized EFSM model are used to extend the model with either more control locations or more location variables.

In our approach, we first observe the behavior of the protocol entity when the parameters of input messages are from a small domain. We enforce a restriction on the minimal size of the domain, so that all occurring test for equality between the parameters can be calculated from the observations. Using the regular inference algorithm by Niese [Nie03] (which adapts Angluin's L^* algorithm to Mealy machines), we generate a finite-state Mealy machine, which describes the behavior of the component on this small domain. We thereafter fold this finite-state Mealy machine into a smaller EFSM model.

Folding the Mealy machine into an EFSM model is performed in four steps. In the first step the algorithm calculates for each state which data values must be remembered by a corresponding EFSM model in order to produce its future behavior. These data values are the basis for constructing the location variables required in the corresponding control location of the EFSM model. In the second step, we use the data values inferred in the first step to transform transitions in the Mealy machine into a so called symbolic normal form, which is designed to capture exactly the equalities and inequalities between symbolic parameters and location variables. In the third step, we merge states of the Mealy machine into locations in the EFSM, if the symbolic forms of their future behavior are the same, using an adaptation of a standard partition-refinement algorithm. In the fourth and final step, we transform transitions from symbolic normal form to the standard form in the EFSMs, and merge transitions when possible.

Discussion

In our strive to model entities of communication protocols we continue the work in the area of domain-specific optimization for regular inference. The approach presented in this paper makes use of knowledge about what tests occur for a type of parameters. It also takes one step closer to modeling natural models of communication protocols, introducing state variables in the model.

The number of membership queries required to be asked by the algorithm is exponentially bounded by the number of state variables and the number of parameters of the input PDU types. However, we think that an addition of a symmetry filter, which would filter out membership queries which have the same differences and equalities between parameter values would make the approach more applicable in practice.

Both the approach presented in Paper III and this paper use the idea that counterexamples guide the approach to which parameter values are necessary to take into account in the model inference. By utilizing this idea, we make it possible to construct models requiring fewer membership queries.

Paper V: Regular Inference for Communication Protocol Entities

In Paper V [BJ08] we continue our work to use regular inference to generate models of communication protocol entities. As in Paper IV, we assume that communication protocol entities communicate via messages consisting of a PDU type and a number of parameters, and that parameters from input can be stored in state variables of communication protocols for later use. In this work we concentrate on parameters that have finite domains. A communication protocol may be designed such that its behavior is structured into control states. Our primary goal is to infer a model such that its control locations are similar to the control states of a natural model of the communication protocol. However, this is not easy since control states are not externally observable, and there are many ways in which to structure the behavior of a communication protocol entity into control locations.

Our motivation is to infer a model of an executable specification of the Advanced Mobile Location Center (A-MLC) protocol developed by Mobile Arts. The protocol is a product that allows Mobile Network Operators to provide presence information from the GSM/UMTS network. A-MLC is commercially available and has been deployed at several telecom operators within Europe. The protocol is implemented in Erlang, a programming language developed by the telecom company Ericsson. The originators of the A-MLC protocol have written a functional specification of the protocol [BJ03], from which an executable specification can be generated. We were allowed access to an executable specification, and performed experiments with it. We based our evaluation of our approach on the results from the experiments.

Our approach infers EFSM models. Each transition in the EFSM model is labelled with three parts. The first part consist of a PDU type and a number of symbolic parameters, representing input. The second part is an assignment of symbolic parameters to location variables. The third part consist of a PDU type and a number of concrete parameter values, representing output. The EFSM model has location variables, in which input parameter values can be stored for later use. Our approach intends to infer control locations that are similar to the control states of the executable specification. It does so, based on an observation in the executable specification that, whenever executing in a control state on an input message with a certain PDU type and outputting a message with a certain PDU type, then the next control state is usually the same. Thus, a sequence of input and output PDU type pairs, uniquely determines a control state in the executable specification. As in Paper IV, the approach has two phases; we first use an existing regular inference technique to infer a finite state Mealy machine, and thereafter fold it into an EFSM.

We have used LearnLib to infer a Mealy machine model of the executable specification of the A-MLC protocol. The result is a Mealy machine with 43 states and 1560 input symbols. Since the Mealy machine is so small we were able to estimate a set of twelve control locations for the EFSM. The executable specification contains thirteen control states. Out of in total twelve estimated control locations we were able to match nine control locations to each a single control state, two pairs of control locations to each one control state, and one control location to two different control states in the executable specification.

Discussion

In this paper we have continued the work started in Paper IV, inferring models with location variables taking input messages with parameter values that may be stored in location variables, and used in guards to decide future behavior or be used in output messages. However, a difference is that in this work we assume that parameter values are from finite, small domains. We have "shrunk" originally large domains by choosing a small amount of representative values from the original domains.

We think it would be interesting to add more data values to the domains of the input parameters and rerun LearnLib on the executable specification with the new input alphabet. This would most likely result in a Mealy machine that includes more behavior of the executable specification. Hopefully an EFSM model of this Mealy machine would be more similar to the executable specification and contain more transitions.

4. Related Work

In this chapter we review work on regular inference to infer models or specifications of systems, and work done to optimize the technique. We structure the chapter according to the intended application.

Regular Inference in Model Checking

Several researchers have combined regular inference and model checking to be able to model check SUTs for which no model exists a priori, and for which no source code is available, so called black boxes. A model of the SUT can then be constructed using regular inference. Peled, Vardi, and Yannakakis [PVY99] present approaches that model check a system given a specification but without an existing model of the system. One of their approaches uses regular inference to construct a model, to which model checking is applied. An overview of this approach is presented in Section 2.8. They combine regular inference and model checking algorithms so that spurious counterexamples, i.e., counterexamples generated by the model checking algorithm which do not correspond to possible executions of the SUT, are fed back to the regular inference algorithm and used to refine the model. They have also presented an extension to their approach, called Adaptive Model Checking, which makes use of an existing incorrect (but not irrelevant) model of the SUT [GPY02]. In this approach they extract from the existing model two sets of strings, with which they initialize the set of prefixes S and suffixes E, in the observation table $\mathcal{OT}$, which the L^* algorithm maintains. The L^* algorithm is presented in detail in Section 2.1.

Assume we want to do model checking on a system for which we are supplied with a set of initial states, and a transition function by which we can compute the set of states reachable in one step from any given set of states. To perform model checking we iterate the function to find a fixpoint where we have calculated all reachable states. A problem is that, in general, such a fixpoint may not be computable in a finite number of iterations. Therefore various so-called widening or acceleration techniques have been developed, by which the fix point is guessed from information of a small number of its approximations. If the state is a string over a finite alphabet this guessing can be performed using regular inference. One context in which this has been tried is regular model checking. Regular model checking is a method for verifying systems consisting of an arbitrary number of homogeneous finite-state processes connected in a ring topology [BJNT00]. The systems have their configurations coded as strings over a finite alphabet, sets of configurations as finite automata, and transitions as finite transducers.

Habermehl and Vojnar [HV04] have presented an approach to perform model checking on such a system, using the regular inference algorithm by Trakhtenbrot and Barzdin [TB73] to infer all reachable configurations in the system. The Trakhtenbrot and Barzdin algorithm requires answers to membership queries for all strings up to some certain length k. The approach by Habermehl and Vojnar uses the transition function of the system to compute answers to membership queries. The answer is $+$ for all strings of length k that leads to a configuration of the system, all others of length k are answered $-$. There are no equivalence queries in the sense used by Angluin's L^* algorithm. Instead the approach checks if the transition function has reached a fixpoint; if not, it does membership queries for all strings of one size longer than queried for before, and continues the inference. The approach focuses on model checking whether or not a set of configurations conform with the specification. This is computed by checking whether or not the intersection between the set of configurations not conforming with the specification, and the set of reachable configurations is empty.

Vardhan and Viswanathan and others [VSVA04, VV06] present a similar approach, which uses either the regular inference algorithm by Kearns and Vazirani [KV94] or the RPNI algorithm [OG92, Dup96] to infer a regular set of reachable configurations. This approach also calculates membership queries from the transition function, and does not use equivalence queries. The answer to a membership query is $+$ for strings that lead to configurations of the system, and $-$ for strings for which the transition function is not defined. Once the approach has inferred a DFA model it performs model checking on it. Counterexamples returned from the model checking procedure is run in the system to find out if they are real counterexamples or spurious. Spurious counterexamples are feed back as negative counterexamples to the inference algorithm. If no counterexample is returned from the model checking procedure the approach checks that the transition relation has reached a fixpoint. The approach reports that the system conforms with the specification if a fixpoint has been reached, otherwise the inference of reachable configurations must continue.

Regular Inference in Compositional Verification

Regular inference and model checking have also been combined to verify systems that can be partitioned into several components. Compositional verification can be used to verify concurrent components in a system that cannot be handled as a large single system within the time or space limits of existing verification techniques. The simplest idea of compositional verification, e.g., in the case of two components, is to regard each component as the environment of the other. The first step is to construct a model of the environment of one component, which includes the other component, such that the component acts without violating the specification of the system. The second step is then to check that the other component has no other behavior than specified in the model of the environment.

Cobleigh, Giannakopoulou, and Pasareanu present the first approach that uses regular inference to infer a model of a component's environment, which then is used in compositional verification [CGP03]. They assume that a system consists of two components. [1] Components synchronize on symbols common to their alphabets and interleave the remaining actions. The approach has access to models of the components and the specification of the system. In a first step, Angluin's L^* algorithm infers an environment model of one of the components under which it conforms with the specification. The environment model is inferred by letting the answer on a membership query for a string be $+$, if the string is accepted by the system consisting of the component and the specification of the system. An equivalence query is answered *yes* if model checking concludes that the environment does not accept other strings than the other component does.

Chaki and Strichman have extended this work to an approach that aims to reduce the number of required membership queries used by the regular inference algorithm to infer a model of the environment [CS07, CS08]. The idea is to use a small alphabet in order to ask few membership queries. Their approach starts with the empty alphabet, and extends it whenever a counterexample indicates that it is necessary. This means that they infer a model of as small part of the interface as possible. However, finding a minimal alphabet is difficult so their practical approach is to formulate their search into a constraint problem to be solved. The approach of refining the alphabet is also presented by Pasareanu et al. [PGB$^+$08], and they present several heuristics for how to extend the alphabet. Sinha and Clarke present an approach [SC07], in which the two components communicate via shared memory, i.e., they share a set of variables. Hence, the alphabet is exponential in the number of shared variables. They avoid the problem of using a large alphabet in regular inference, by clustering symbols of the alphabet in each state of the model. A symbolic model checker answers both membership and equivalence queries. They partition a cluster of symbols whenever a counterexample indicates that is required. They have also extended their approach [CCSS08] to using predicate abstraction [CU98, GS97] on the variables. Nam, Madhusudan, and Alur also present an approach for compositional verification of memory sharing components [NMA08], which communicate via shared boolean variables. They repartition the system into a set of components in a way so that the number of shared variables are small. They use a BDD-based implementation of L^* [AMN05, NA06].

[1] They also explain how their approach can be extended to systems consisting of a finite number of components.

Optimizations for Reactive Systems and Systems with Certain Alphabet Properties

Hungar, Niese, and Steffen have presented domain-specific optimizations in regular inference, which aims at reducing the number of membership queries required to infer a model of the SUT [HNS03]. They present among others separate optimizations for systems with prefix-closed languages, and systems with languages that have independent and symmetric symbols. The latter optimization requires that an expert specifies independence relations over symbols. The two optimization for prefix-closed systems on a small set of examples gave an average reduction of membership queries of 74%. In Paper II we applied one of two optimizations for prefix-closed systems to protocols shipped with the Edinburgh Concurrency Workbench [MS], and attained almost the same level of reduction: on average 59%.

Parameterized Systems

Regular inference is being used to generate models of communication protocol entities. These type of systems communicate via messages consisting of a PDU type and a number of parameters, which typically involves a large amount of symbols in the input alphabet used by the inference algorithm.

Hagerer, Hungar, Niese, and Steffen have presented an approach that uses regular inference to construct finite automata with symbolic alphabets of SUTs [HHNS02]. Their idea is to first generate strings observed in the SUT, then manually remove parameters or symbols that the user thinks are uninteresting, e.g., time stamps. After that, exchange the remaining parameters to symbolic values according to a predefined specification. Next, make an automaton accepting exactly the symbolic strings, and finally merge states that are seemingly equivalent. The automaton model is then validated against expert-provided properties which should hold for the system. They have used their approach to infer a model of a part of a telephone switch, and used their model to monitor the switch to report when errors occur. They have also used their approach for test-suite enhancement [HHM$^+$02].

Li, Groz, and Shahbaz uses regular inference to infer partial models of commercial of the shelf (COTS) components, which they use in their integration testing procedure [LGS06]. They generate tests from the models of the components. These tests may reveal unknown input action types which are output from one component and input to another. They focus on a type of model which takes parameter values into account, since an output from one component input to another component may disclose an incompatibility, which was not discovered when testing the components in isolation. Their approach infers finite state machine models with input and output symbols, each of which consists of an input (or output) action type together with a sequence of parameter values. The model does not have any state variables. Furthermore, they assume that output action type depends only on input action type and not on the parameters. Later, they present an approach inferring a variant of the model, which allows the parameter values in input to affect the output type behavior [SLG07]. Both approaches are based on the regular inference algorithm for Mealy machines presented in Section 2.2. The prefixes in the observation table are strings of input action types and parameter values, and the suffixes are strings of output action types. In the first approach, each entry in the observation table contains pairs of input and output parameter value sequences. In the second approach, each entry contains pairs of input parameter value sequence and sequences of output action type and output parameter values. The approaches use a subset of the complete input alphabet in the regular inference algorithm. The parameter values are selected from tests or chosen randomly. The observation tables must fulfill extra criteria due to the parameter values, which give rise to more membership queries.

Grinchtein, Jonsson, and others, have extend the regular inference technique to timed systems [GJL05, GJL04, GJP06]. They present an approach that infers a model called event-recording automata, in which each symbol consists of an action type and a clock valuation, i.e., a parameter that specifies what time the action type occurred. They infer a symbolic model which has guards on transitions consisting of conjuncts of clock constraints, where a clock constraint is a comparison of type $<$, $\leq$, $>$, or $\geq$, between an event-recording clock and a natural number. Huselius presents a methodology for constructing models of instrumented real-time systems by analysis of execution traces [Hus07].

Specification Mining

Specification mining is a machine learning approach to discovering formal specifications. A model of a well-functioning system captures properties of the system, which can be considered as a specification of the system. In this setting it is commonly assumed that interactions with the SUT are only monitored, i.e., membership queries can not be asked, and therefore only correct behavior of the SUT is observed. Correct behavior of the SUT correspond to accepted strings in the regular inference algorithm. An inference technique that can be used for this situation is the k-tails algorithm by Biermann and Feldman [BF72]. The algorithm first constructs an automaton accepting exactly the strings that have been collected in the monitoring phase. Next, it merges states that accept the same set of strings up to some length k, and this repeats until no more merges are possible.

Ammons, Bodik, and Larus present an approach that infers application program interface (API), or abstract data-type (ADT) specifications of existing programs that are working almost correctly [ABL02]. Their approach infers nondeterministic finite automata models with edges labelled by interactions in an abstract form, from sequences of interactions with API/ADT interfaces.

Before applying the inference algorithm several steps are taken to transform the strings of interactions to an abstract form. They replace parameter values with symbolic names in the strings, and then apply the inference algorithm to the resulting strings. Their approach uses an existing inference algorithm by Raman and Patrick [RP97], based on the Biermann and Feldman algorithm [BF72], which constructs probabilistic finite state automata (PFSA) with edges, each labelled with a symbol and a weight representing the frequency of that edge. The inference algorithm by Raman and Patrick merges states that accept the same set of k-length strings with weights higher than some certain threshold. The approach considers infrequent behavior as "uninteresting". Therefore edges that are not likely to be traversed while generating a string from the PFSA are removed. In the final step, the weights on the transitions are removed, and the result is a non-deterministic finite automaton.

Lorenzoli, Mariani, and Pezzè have presented an approach to infer models of method invocation call sequences of components [LMP06]. The inferred model is an extended finite state automaton, in which the alphabet is the finite set of methods that can be invoked. They collect traces by observing the system under execution. Their approach first merges traces which have the same sequences of method invocations. Next they generate invariants for each method invocation in the sequence over the parameters for the method invocation with the tool Daikon, which dynamically discover invariants over variables [ECGN01]. The approach then applies the Biermann and Feldman algorithm [BF72] to the collected traces. They present different senses of state equivalence relations, which are used in the merging process of the Biermann and Feldman algorithm. Finally they transform the automaton into their model. In later work [LMP08] they extend their approach to allow variables associated to methods, for instance state variables in the SUT. This approach infers models in a similar way to previous work, from sequences of tuples, each tuple consisting of a method invocation, a domain for an input parameter, and a value for a variable associated to a method. In other work by Mariani, and Pezzè [MP05, MP07], they infer a finite state automata model for sequences of method invocations, and an invariant over the parameter values used in the method invocations, produced by Daikon.

Alur et al. [AvMN05] present an approach to infer interface specifications for Java classes. Given a set of unsafe, i.e., undesirable evaluations of class variables for a class interface, the approach checks if the specification can reach a state in which the class variables have unsafe evaluations. They first apply the predicate abstraction technique [CU98, GS97] to the specification, then they use regular inference to infer an environment for the abstract specification, which does not put the abstract specification in a state in which the class variables have unsafe evaluations. Their approach uses answers from a model checker, applied to the inferred model and the unsafe variable evaluations, to compute membership and equivalence queries. To conclude that the answer to an equivalence query is *yes*, they use approximate and heuristics.

Besides state machines, Message Sequence Charts (MSCs) [IT99] have been used to specify communication protocols. Alur, Etessami, and Yannakakis [AEY03] have presented an approach in which the designers of protocols specify parts of the protocol in MSCs, and then regular inference is used to discover which other parts of the protocol needs to be specified in order to conclude that the specification is deadlock-free.

Tools

Raffelt and Steffen provide a tool for regular inference called Learn-Lib [RSB05, RS06]. The tool can infer DFAs and Mealy machines, and provides optimizations for inference of DFAs to reduce the number of required membership queries. There are optimizations which make use of system-expert specified independencies and symmetries between symbols, and optimizations for prefix-closed SUTs. The tool also provide means to perform equivalence queries by implementing conformance tests: the W- [Cho78], and Wp-Method [FvBK$^+$91], or sets of randomly generated tests.

5. Conclusions and Future Work

5.1 Conclusions

In this thesis we have presented work that investigate and adapt regular inference techniques for communication protocol entities. This thesis focuses on two problems: regular inference techniques requires a large amount of membership queries when inferring models of communication protocol entities, and the inferred models of communication protocol entities are large.

We have in Paper I surveyed three algorithms for regular inference, presented their similarities and their difference in terms of upper bounds on the required number of membership queries. For one of the surveyed algorithms, we have in Paper II compared how it performs on different types of DFAs and related it to the theoretical upper bound on the number of required membership queries.

In Paper II we present the result of investigating how many membership queries the well-established regular inference L^* algorithm by Angluin requires for different type of DFAs. The algorithm has been investigated on randomly generated examples and prefix-closed DFAs, which can be used to model reactive systems. Our results show that, inferring a prefix-closed DFA model requires in general more membership queries than are required for randomly generated DFA models. This is a negative result for our attempt to infer models of communication protocol entities. However, with an optimization for prefix-closed DFAs we were able to reduce the required number of membership queries with about 20%.

The number of input messages to a communication protocol entity is exponential in the number of parameters and this negatively affects the required amount of membership queries. We have have presented two approaches that aim at reducing the required number of membership queries, and infer symbolic EFSM models. The approaches have different assumptions on the domain of the parameters. The approach in Paper III infers symbolic EFSM models without location variables assuming that parameter values are boolean, and the approach in Paper IV infers symbolic EFSM models with location variables assuming parameter values are from a very large or infinite domain. The approach in Paper III avoids asking membership queries for a PDU type and parameter settings that do not affect the behavior in a state. The approach in Paper IV use a small subset of the very large domain for parameters in membership queries.

In Paper V we have presented an approach that infers symbolic EFSM models with control locations which are similar to the control states of natural models of communication protocol entities. In the construction of the approach in Paper V we were guided by an executable specification of the Advanced Mobile Location Center protocol provided by Mobile Arts [BJ03]. The approach in Paper V exploit a common property of the control states in the executable specification to construct a symbolic EFSM model. In this approach we assume that parameters take on values from small finite domains. In contrast to the approach presented in Paper IV, the location variables in this approach are not inferred, instead there is a location variable for each input parameter. Our experimental results show that the approach can successfully exploit the property of the control states to construct similar control locations in a symbolic EFSM model.

Common regular inference techniques infer large DFA or Mealy machine models of communication protocol entities. We have handled this problem by inferring compact symbolic EFSM models. In the approach in Paper III we simultaneously infer the symbolic transitions and construct the observation table. In the approaches in Paper IV and Paper V we first infer a Mealy machine model and then fold it into a symbolic model. In compact and natural models it is easier to discover existing correlations between parameters and the behavior of the model.

5.2 Future Work

In this section we discuss what further work can be done in order to improve the presented regular inference approaches. We also discuss general ideas on how to enable regular inference of communication protocol entities.

We believe that a similar optimization to the prefix-closure optimization for DFAs investigated in Paper II, can be used in regular inference of Mealy machines. The optimization would be applicable to communication protocol entities which have an error state and remains in that state, i.e., a sink. The optimization would reply "error" to membership queries for extensions of a string that has already been answered "error". It would be interesting to investigate how much less membership queries the use of this optimization for Mealy machines with sinks would require.

The approach presented in Paper V infers symbolic Mealy machines with input symbols containing symbolic parameters, and output symbols containing concrete parameter values. Because of the concrete output parameter values the approach in Paper V may infer unnecessarily large result expressions on transitions. One way to handle this is to extend this approach so that the parameters in output symbols are symbolic as well. An idea is to first calculate which output parameter values are parameter values from input, and which are constants, i.e., not parameters from input. Next, we could use ideas from the approach in Paper IV to calculate which parameter values from input need to be stored in states for later use in output and in guards on transitions. Thereafter we could perform a transformation to symbolic transitions based on the calculated values to be stored.

We also have a different suggestion on how to handle the large result expressions on transitions. The approach in Paper V can be improved by using heuristics to construct compact result expression, i.e., to use less formal parameters and location variables in result expressions. We think it is possible to use the same heuristics as used in Binary Decision Diagram packages. By trying out different orders on the location variables and formal parameters a local minimum on the required number of variables and parameters can be discovered. This would imply that we can remove location variables or formal parameters that are not needed in result expressions.

Furthermore, we would like to alter the approach so that it instead of switch statements in result expressions we input operators from users, and use the operators to construct more compact result expressions. Examples are binary operators such as $\neq$, $<$, and other operators commonly used in communication protocol entities, e.g., a check whether a parameter is defined. Using the appropriate operator for a parameter in the model would make the labels on transitions more compact.

In the approaches presented in Paper IV and Paper V, we infer Mealy machines which we fold into compact symbolic models. We think it would save computation time and space if the approaches would not create the larger intermediate model. Instead we would like them to be more similar to the approach in Paper III, in which we directly calculate the guards that will label the transitions in the corresponding symbolic model. Inferring equivalence classes of parameters, based on some operator on the parameter values, may save membership queries, since membership queries essentially need to be asked for one representative of each equivalence class. However, the result in Paper III most likely also apply to this setting: the number of membership queries are only reduced for entities in which a small part of the parameters affect the behavior of the entity.

In the future it is interesting to further investigate how well the approach presented in Paper V applies to other communication protocols than the protocol we used in our experiments. This would probably give us new insights to how to refine our approach. For instance, we may get insights to better or alternative ways of partitioning the states in the Mealy machine into control locations. This may lead to new approaches for constructing control locations.

In order to facilitate the use of regular inference applied to communication protocol entities, it is desirable to construct a tool which combines the approaches we have presented for different types of parameters. A user should be able to specify the domains of the input parameters and appropriate operations on them in the tool, information which the tool uses to select the appropriate approach to apply to each parameter. We also think that approaches to automatically calculate or estimate parameter operations must in the long run be developed, in order for such a tool to be fully automatic. Giving the user the ability to indicate which parameters that are likely to be used in guards on transitions or in output, could be a way of speeding up the inference. This information would guide the inference algorithm towards which parameter values may need to be stored in state variables, and towards a more user friendly model.

Bibliography

[ABL02] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, pages 4–16, 2002.

[AEY03] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Trans. Softw. Eng.*, 29(7):623–633, 2003.

[AMN05] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[AvMN05] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109, New York, NY, USA, 2005. ACM.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14(1), Jan. 1989.

[BDG97] J.L. Balcázar, J. DÃaz, and R. Gavaldá. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Kluwer, 1997.

[BF72] A. Biermann and J. FELDMAN. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.

[BGJ$^+$05] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.

[BH06] Luciano Baresi and Reiko Heckel, editors. *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*. Springer, 2006.

[BHS91] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with applications from protocol specification*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[BJ03] J. Blom and B. Jonsson. Automated test generation for industrial erlang applications. In *Proc. 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14, Uppsala, Sweden, Aug. 2003.

[BJ08] Therese Bohlin and Bengt Jonsson. Regular inference for communication protocol entities. Technical Report 2008-024, Department of Information Technology, Uppsala University, September 2008.

[BJK$^+$04] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[BJLS05] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. *Electr. Notes Theor. Comput. Sci.*, 118:3–18, 2005.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer Verlag, 2000.

[BJR06] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In Baresi and Heckel [BH06], pages 107–121.

[BJR08] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001*, pages 203–213, 2001.

[BR04] Therese Berg and Harald Raffelt. Model checking. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander

Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 557–603. Springer, 2004.

[CCSS08] Sagar Chaki, Edmund M. Clarke, Natasha Sharygina, and Nishant Sinha. Verification of evolving software via component substitutability analysis. *Formal Methods in System Design*, 32(3):235–266, 2008.

[CDH$^+$00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. 22nd Int. Conf. on Software Engineering*, June 2000.

[CGP03] J.M. Cobleigh, D. Giannakopoulou, and C.S. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03, 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Verlag, 2003.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering*, 4(3):178–187, May 1978. Special collection based on COMPSAC.

[CS07] Sagar Chaki and Ofer Strichman. Optimized l*-based assume-guarantee reasoning. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2007.

[CS08] Sagar Chaki and Ofer Strichman. Three optimizations for assume-guarantee reasoning with l^*. *Formal Methods in System Design*, 32(3):267–284, 2008.

[CU98] M.A. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer Verlag, 1998.

[Dup96] P. Dupont. Incremental regular inference. In L. Miclet and Colin de la Higuera, editors, *ICGI*, volume 1147 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 1996.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[FvBK+91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. on Software Engineering*, 17(6):591–603, June 1991.

[GJL04] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. In Yassine Lakhnech and Sergio Yovine, editors, *FORMATS/FTRTFT*, volume 3253 of *Lecture Notes in Computer Science*, pages 379–396. Springer, 2004.

[GJL05] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Inference of timed transition systems. *Electr. Notes Theor. Comput. Sci.*, 138(3):87–99, 2005.

[GJP06] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2006.

[Gol67] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[GPY02] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02, 8^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2002.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[Har87] D. Harel. StateCharts : A visual approach to complex systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[HHM+02] Andreas Hagerer, Hardi Hungar, Tiziana Margaria, Oliver Niese, Bernhard Steffen, and Hans-Dieter Ide. Demonstration of an operational procedure for the model-based testing of cti systems. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 336–340, London, UK, 2002. Springer-Verlag.

[HHNS02] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02, 5^{th} Int. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.

[HNS03] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, 2003.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[Hus07] Joel Huselius. *Reverse Engineering of Legacy Real-Time Systems: An Automated Approach Based on Execution-Time Recording.* PhD thesis, Mälardalen University, June 2007.

[HV04] Peter Habermehl and Tomas Vojnar. Regular model checking using inference of regular languages. In *Proceedings of 6th International Workshop on Verification of Infinite-State Systems – INFINITY 2004*, pages 61–71, 2004.

[ISO89a] ISO. Estelle: A formal description technique based on an extended state transition model. Technical Report ISO 9074, International Standards Organization, Geneva, Switzerland, 1989.

[ISO89b] ISO. Lotos – a formal description technique based on the temporal ordering of observational behaviour. Technical Report ISO 8807, International Standards Organization, Geneva, Switzerland, 1989.

[IT99] ITU-T. Recommendation Z.120, Message Sequence Charts. In *Languages and general software aspects for telecommunication systems*, Geneva, Nov. 1999.

[KV94] M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[LGS06] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450, 2006.

[LJL+07] Jae-Dong Lee, Jae-Il Jung, Jae-Ho Lee, Jong-Gyu Hwang, Jin-Ho Hwang, and Sung-Un Kim. Verification and conformance test generation of communication protocol for railway signaling systems. *Comput. Stand. Interfaces*, 29(2):143–151, 2007.

[LMP06] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Inferring state-based behavior models. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 25–32, New York, NY, USA, 2006. ACM.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *30th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2008.

[LS95] Pat Langley and Herbert A. Simon. Applications of machine learning and rule induction. *Communications of the ACM*, 38:55–64, 1995.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, 84(8):1090–1126, 1996.

[Mil80] R. Milner. A calculus of communication systems, 1980.

[MMP$^+$95] O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program amore. Technical Report 9507, Inst. f. Informatik u. Prakt. Math., CAU Kiel, 1995.

[MP05] Leonardo Mariani and Mauro PezzÃ¨. Behavior capture and test: Automated analysis of component integration. In *proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[MP07] Leonardo Mariani and Mauro Pezzè. Dynamic detection of cots component incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[MS] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://homepages.inf.ed.ac.uk/perdita/cwb/.

[NA06] Wonhong Nam and Rajeev Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In Susanne Graf and Wenhui Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 170–185. Springer, 2006.

[Nie03] Oliver Niese. An integrated approach to testing complex systems. Technical report, Dortmund University, 2003. Dissertation.

[NMA08] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Form. Methods Syst. Des.*, 32(3):207–234, 2008.

[OG92] J. Oncina and P. García. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, 1992.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.

[PGB$^+$08] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[PVY99] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV*, pages 225–240, Beijing, China, 1999. Kluwer.

[RP97] Anand Raman and Jon Patrick. The sk-strings method for inferring pfsa. In *In Proceedings 14th International Conference on Machine Learning-ICML'97, Nashville, Tennessee.*, 1997.

[RS89] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences (extended abstract). In *STOC*, pages 411–420. ACM, 1989.

[RS93] R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.

[RS06] Harald Raffelt and Bernhard Steffen. Learnlib: A library for automata learning and experimentation. In Baresi and Heckel [BH06], pages 377–380.

[RSB05] H. Raffelt, B. Steffen, and T. Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, New York, NY, USA, 2005. ACM Press.

[SC07] Nishant Sinha and Edmund M. Clarke. Sat-based compositional verification using lazy learning. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2007.

[SLG07] M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2007.

[TB73] B.A. Trakhtenbrot and J.M. Barzdin. *Finite automata: behaviour and synthesis*. North-Holland, 1973.

[Vas73] M. P. Vasilevski. Failure diagnosis of automata. *Cybernetic*, 9(4):653–665, 1973.

[VSVA04] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Learning to verify safety properties. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004.

[VV06] Abhay Vardhan and Mahesh Viswanathan. Lever: A tool for learning based verification. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 471–474. Springer, 2006.

[WRL94] Bernard Widrow, David E. Rumelhart, and Michael A. Lehr. Neural networks: applications in industry, business and science. *Commun. ACM*, 37(3):93–105, 1994.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 605

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-9559

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2009