Further development of learning techniques

Antonia Bertolino, Antonello Calabro, Sofia Cassel, Yu-Fang Chen, Falk Howar, Bengt Jonsson, Maik Merten, Antonino Sabetta, B. Steffen

CONNECT

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D4.2

Further development of learning techniques

SEVENTH FRAMEWORK PROGRAMME

http://www.connect-forever.eu

INRIA docomo DOCOMO Euro-Labs LANCASTER UNIVERSITY THALES Universita' dell'Aquila technische universität dortmund UNIVERSITY OF OXFORD UPPSALA UNIVERSITET

Project Number	:	231167
Project Title	:	CONNECT – Emergent Connectors for Eternal Software Intensive Networked Systems
Deliverable Type	:	R

Deliverable Number	:	D4.2
Title of Deliverable	:	Further development of learning techniques
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	1.2
Contractual Delivery Date	:	31 Jan. 2011
Actual Delivery Date	:	17 Feb. 2011
Contributing WPs	:	WP4
Editor(s)	:	Bengt Jonsson (UU), Bernhard Steffen (TUDo)
Author(s)	:	Antonia Bertolino (CNR), Antonello Calabrò (CNR), Sofia Cassel (UU), Yu-Fang Chen (UU), Falk Howar (TUDo), Bengt Jonsson (UU), Maik Merten (TUDo), Antonino Sabetta (CNR), Bernhard Steffen (TUDo)
Reviewer(s)	:	Amel Bennaceur (INRIA), Valérie Issarny (INRIA), Massimo Tivoli (UDA)

Abstract

The CONNECT *Integrated Project aims at enabling continuous composition of networked systems, by developing techniques for synthesizing connectors. A prerequisite for synthesis is to learn about the interaction behavior of networked peers. The role of WP4 is to develop techniques for learning models of networked peers and middleware through exploratory interaction.*

During Y1 of CONNECT*, exploratory work was performed to understand the requirements for learning techniques in the* CONNECT *process, and to develop concepts for using* a priori *knowledge about component interfaces as a basis for learning. During Y2, a major goal has been to develop techniques for automatically learning models of networked peers and middleware, based on the concepts developed during Y1 (cf. D4.1). This deliverable surveys the significant progress made on problems that are important for realizing this goal. We have developed techniques for drastically improving the efficiency of active learning, meaning to explore significantly larger parts of component behavior within a given time. We verified the power of our approaches by winning the ZULU challenge in competition with several very strong groups in the language learning community. We have also made significant breakthroughs for learning of rich models with data. We have developed a novel compact and intuitive automaton model for representing learned behavior in a canonical way. Canonicity is very helpful for organizing the resuls of exploratory interactions, since it allows the extension of stable techniques for learning classical finite-state automata (such as L^*) to richer models with data. We confirm this by using the new automaton as a basis for an extension of L^* to richer automaton models. During Y1, we have identified abstractions as an important concept in the process of learning behavioral models of realistic systems: we introduce a method for refining a given abstraction when needed during the learning process. We also present a specialization of learning, which does not generate complete models of behavior, but concentrates on aspects that are relevant in a specific context. Finally, the deliverable reports on our efforts to learn non-functional properties, the development of a monitoring infrastructure, and further development of the learning tool infrastructure, based on LearnLib.*

Document History

Version	Type of Change	Author(s)
1.0 1.1 1.2	Initial version for internal review Revised version including first review feedback Version after internal review	All All All

Document Review

Date	Version	Reviewer	Comment
18.01.2011 10.02.2011	1.0 1.1	Massimo Tivoli, Amel Bennaceur Valérie Issarny	– –

Table of Contents

List of Figures

List of Tables

1 Introduction

The CONNECT Integrated Project aims at enabling continuous composition of networked systems to respond to the evolution of functionalities provided to and required from the networked environment. In this context, being able to automatically learn the behavior of networked components is vital.

At present the efficacy of integrating and composing networked systems depends on the level of interoperability of the systems' underlying technologies. However, interoperable middleware cannot cover the ever growing heterogeneity dimensions of the networked environment.

CONNECT aims at dropping the interoperability barrier by adopting a revolutionary approach to the seamless networking of digital systems, that is, synthesizing on the fly the connectors via which networked systems communicate. Connectors are implemented through a comprehensive dynamic process based on (i) extracting knowledge from, (ii) learning about and (iii) reasoning about, the interaction behavior of networked systems, together with (iv) synthesizing new interaction behaviors out of the ones exhibited by the systems to be made interoperable, and further (v) generating and deploying corresponding connector implementations.

This aim raises challenges for modeling and reasoning about system and connector behaviors, and for synthesizing specifications of connector behavior. A high level view of CONNECT operation is described in Section 5.1 of D1.1[1], as a system of various enablers that exchange information about the networked system to be constructed, as shown in Figure 1.1.

Work Package 4 provides CONNECT with the Learning enabler. In the CONNECT scenario, one cannot expect all networked systems to provide formal specifications of their interaction behavior. For this reason, it is then necessary to have learning algorithms and techniques to dynamically infer specifications or models of the connector-related behavior of networked peers and middleware.

In particular, the Learning enabler (represented by the box labeled "WP4: Learning" in Figure 1.1) takes interface descriptions of components in the networked system, along with information on used data domains, to create a formal model of the behavior of networked systems using exploratory interaction, i.e., analyzing the messages exchanged with the environment. The resulting formal model can be in the form of a Mealy machine or a Labeled Transition System (LTS).

A major challenge for WP4 in CONNECT is to develop techniques for learning rich models of components in networked systems. Such models will describe both control and data aspects of a component's behavior, and also cover non-functional aspects. This will be performed by significantly advancing the state-of-the-art techniques for active learning of behavioral models that capture control and data aspects. Furthermore, the learning enabler should be able to introduce non-functional properties into models, using information about metrics that are measurable when interacting with the networked systems. This can be done, partly by cooperating with the monitoring system, developed in WP5 (see D5.1), which can be used to get information about the (in-)correctness of inferred models, and about observed metrics.

1.1 The role of Work Package 4

It is the task of Work Package 4 to develop techniques to realize the Learning enabler, i.e., to develop techniques for learning and eliciting representative models of the connector-related behavior of networked peers and middleware through exploratory interaction. The objectives, as stated in the Description of Work (DoW)[2], are:

> '... to develop techniques for learning and eliciting representative models of the connector-related behavior of networked peers and middleware through exploratory interaction, i.e., analyzing the messages exchanged with the environment. Learning may range from listening to instigating messages. In order to perform this task, relevant interface signatures must be

[1] CONNECT Deliverable D1.1,'Initial Connect Architecture'
[2] CONNECT Grant Agreement, Annex I

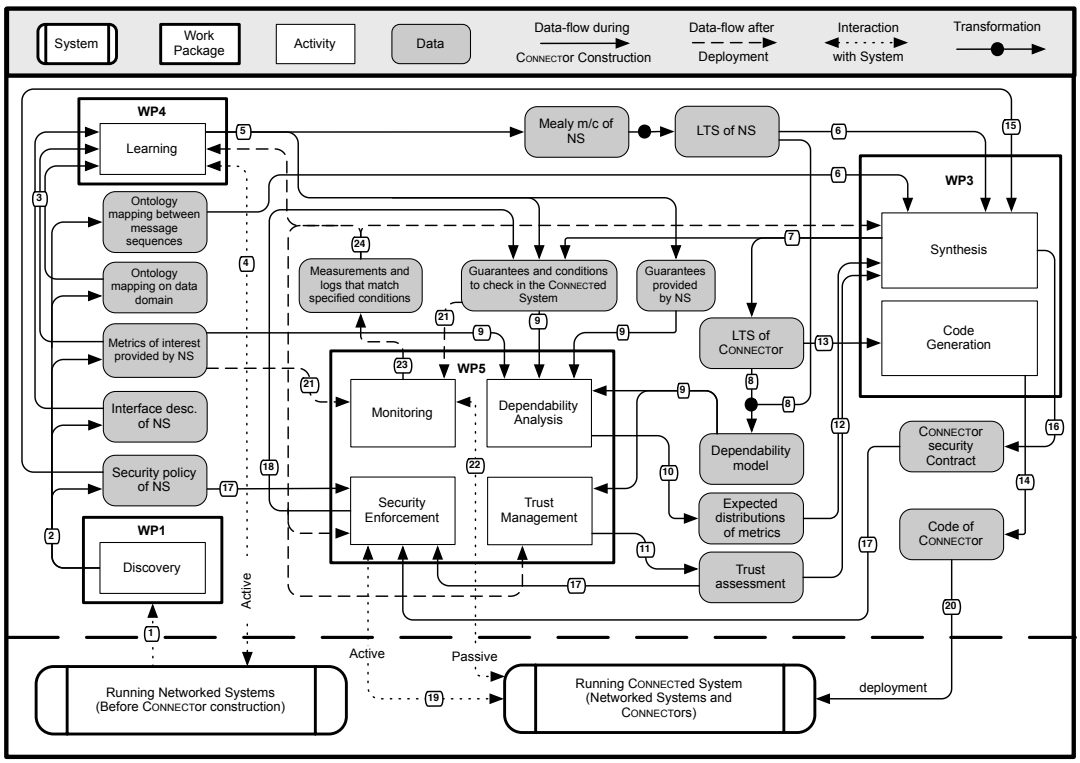

Figure 1.1: Data-Flow in the CONNECT System

available. A bootstrapping mechanism should be developed, based on some reflection mechanism. The work package will investigate minimal requirements on the information about interfaces provided by such a reflection mechanism in order to support the required bootstrapping mechanism. The work package will further support evolution by developing techniques for monitoring communication behavior to detect deviations from learned behavior, in which case the learned models should be revised and adaptors resynthesized accordingly.'

In the DoW, Work Package 4 is structured into three subtasks.

Task 4.1: Learning application-layer and middleware-layer interaction behaviors in which techniques are developed for learning relevant interaction behavior of communicating peers and middleware, and building corresponding behavior models, given interface descriptions that can be assumed present in the CONNECT environment, including at least signature descriptions.

Task 4.2: Run-time monitoring and adaptation of learned models in which techniques are developed for monitoring of relevant behaviors, in order to detect deviations from supplied models.

Task 4.3: Learning tools in which learning tools will be elaborated, by building upon the learning framework developed by TU Dortmund (LearnLib), and considerably extending it to address the demanding needs of CONNECT.

The work in WP4 is based on existing techniques for learning the temporal ordering between a finite set of interaction primitives. Such techniques have been developed for the problem of regular inference (i.e., automata learning), in which a regular set, represented as a finite automaton, is to be constructed from a set of observations of accepted and unaccepted strings. The most efficient such techniques use the setup of *active* learning, where the automaton is learned by *actively* posing two kinds of queries: a *membership query* asks whether a string is in the regular set, and an *equivalence query* compares

a hypothesis automaton with the target regular set for equivalence, in order to determine whether the learning procedure was (already) successfully completed. The typical behavior of a learning algorithm is to start by asking a sequence of membership queries, and gradually build a hypothesized automaton using the obtained answers. When a "stable" hypothesis has been constructed, an equivalence query finds out whether it is equivalent to the target language. If the query is successful, learning has succeeded; otherwise it is resumed by more membership queries until converging at a new hypothesis, etc. A more detailed account of active automata learning is presented in Chapter 2.

1.2 Brief Summary of Achievements in Y1

During Y1, work in CONNECT focused on establishing a basis for addressing the hard challenges for the learning enabler, as described in the Y1 deliverable *D4.1: Establishing basis for learning algorithms*. A large number of case studies were performed. Briefly, the achievements were in the following directions.

Prerequisites for the learning process: From the large number of case studies, we inferred under which conditions learning can be practically conducted, and derived conditions under which the learning enabler will be able to fulfill its role in the CONNECT architecture. In order to learn a model of a component, it is necessary to be able to reset it to a well-defined initial state between successive experiments, and it is assumed that it behaves deterministically at the level of abstraction that is considered.

Handling interface descriptions and data domain information: A major challenge in WP4 is to handle data in the learning enabler. We developed a framework for incorporating information about interface primitives, their parameters, data types, and their ranges, from an interface description, e.g., in WSDL, and for automatically transforming this information into a form usable by the learning tool LearnLib.

Bridging between abstract formal models and concrete system interfaces: The learning enabler must take into consideration that learning can only be performed in terms of concrete interaction with a networked peer, whereas the learning algorithms and the desired output of the learning enabler are in terms of abstract models, e.g., in the form of LTSs. We developed a systematic approach for bridging between these different levels of abstraction, by means of an explicitly added module that concretely performs the translation between different levels of abstraction. This allows to use finite-state learning techniques to generate models of infinite-state modules. This technique was used, e.g., to generate models of (fragments of) the SIP and TCP communication protocols [8], and of the new biometric passport [9]. Another technique was used to learn a model of a rather complex gateway protocol [29].

Tool support: We upgraded LearnLib, our main learning tool, and ported it to the Java platform, ensuring that it is deployable on a range of platforms. Another emphasis was put on the ease of use and intuitive handling of the tool in order to provide it for experimentation within the whole CONNECT consortium.

1.3 Review recommendations

The review for WP4 contained the following passage:

> *The work of the first year is mainly a work plan which has to be carried out in the remaining time of the project.*

Our view is that progress in WP4 during Y2 was achieved in correspondence with the work plan, including to develop techniques for automating learning with data, to initiate work on learning nonfunctional properties, and to push the development and distribution of LearnLib. The review of D4.1also raised the following questions to be considered in future deliverables, which we tried to address in this year's work.

1. *Are meaningful abstractions related to ontologies? And how? What additional input is needed to obtain meaningful abstractions?*

   We have addressed this remark at several levels. In order to produce meaningful abstractions, we have analyzed the semantic requirements of models, leading to different approaches for producing abstractions. Some of these use semantic (ontological) input; some do not:

   Following our roadmap from D4.1, we investigated the prerequisites of abstraction and developed different approaches to abstraction in Y2. As we argue in Chapter 2 and Chapter 7, the learning enabler will have to produce *semantically rich* (i.e., annotated with pre- and postconditions) models in order to enable synthesis of CONNECTors. In Chapter 5, we present a general pattern for learning systems with data. The presented algorithm will construct abstract symbolic models directly. We expect that this pattern can be further improved by semantic (ontological) information about types of parameters and relations between data types.

   On the other hand, in Chapter 6, we present an automated way of producing (behaviourally) meaningful abstractions *without semantic input*. The resulting models will be correct abstractions of a concrete system. However, without additional input, this method will obviously not infer concepts (e.g., not even numbers).

2. *It seems reasonable to learn sequential components with a relatively restricted interface behaviour. Blackbox components may well be of complex dynamic nature with internal concurrency? Can this situation be avoided?*

   This remark points out two problems that regular inference methods in practice have to face. These are (a) the fact that realistic systems, especially systems of systems, are non-deterministic to some extent, and (b) that real systems, consisting of numerous components, may be very complex. We have tried to address both of these points in our work, according to the following:

   (a) Though not addressed explicitly in this deliverable, our pattern for test-drivers (cf. Chapter 8) can be used to hide certain degrees of non-determinism from the learning algorithms. We spent some effort on analyzing the prerequisites for automatic generation of test-drivers and developed a general scheme for the automatic generation of test-drivers (cf. Chapter 2).

   (b) In order do deal with complex, large systems (in terms of the number of states), we improved classic learning algorithms to work with a minimal amount of queries, which is necessary for handling large models (cf. Chapter 3). We also developed a new learning algorithm that infers control behavior and relations on data parameters and internal variables. We expect the algorithm, which will be presented in Chapter 5, to improve on the state-of-the-art wrt. size of models (that is contained information) by at least one order of magnitude.

3. *The tool supposes input complete components and absence of resource related problems. Nevertheless, the usual question of "how to detect quiescence" should occur also in this context.*

   This remark emphasized possible limitations of the learning approach from a tester's perspective; there are certain phenomena that models produced by state-of-the-art regular inference methods do not capture. Considering the restricted scenario of CONNECT, however, we feel that both input completeness and quiescence do not hamper the learning process. Most interaction with networked components is envisioned to happen on the protocol level. Sending an invalid packet to a system will then only lead to an increased amount of tests, not to failure of the system.

   Wrt. quiescence, in D4.1 we introduced and discussed the usual assumption that networked components have 'stable' states from which no further output occurs and input is enabled. Regular inference will (for now) only deal with stable states. However, we will investigate ways of relaxing this assumption in Y3, especially when learning non-functional properties, such as reliability. The design pattern for test-drivers presented in Chapter 8 could be the basis for extending learning beyond stable states.

4. *It is also supposed that components can be accessed directly and there is no interference between component and network properties. Is this realistic?*

The focus of this remark is the potential pitfalls of the practical application of a theoretic approach. We adressed this remark in two ways, referring to Chapter 8 for details. On the one hand, as with most of the other remarks, we developed a method for "hiding" reality from the regular inference algorithms. On the other hand, we also spent some effort of learning non-functional properties, such as latency and reliability, and making this explicit in the learning process.

The following sections provide an overview on the challenges and achievements.

1.4 Challenges for Y2

During Y2, the main challenge has been to develop techniques for automated learning of rich component models, based on the findings during Y1. The overall goal has been to make the conceptual breakthroughs that will enable the learning enabler to fulfill its purpose in the CONNECT architecture. The challenges that have been addressed include the following.

Extending automated automata learning to rich models State of the art learning algorithms work on rather unexpressive finite-state models. To support all the phenomena that are typically used in the specification of networked components (parameterized actions, functional and non-functional parameters, state- or global variables) in a feasible fashion, the underlying models will have to be extended accordingly.

Thus, the challenge is to extend learning techniques to handle richer models including data of various form. Within a suitable general framework, we should develop appropriate specializations to handle learning for specific types of systems. Examples of such classes include communication protocols, which require a proper treatment of parameters, and web services, where associated metadata descriptions should be exploited by suitable techniques. These specializations should result in modules of learning tools.

Handling QoS. The CONNECT framework also takes QoS parameters into account. We will develop techniques for monitoring and learning QoS properties of interaction behavior. A first approach should cover latency behaviors. A satisfactory incorporation of QoS requires, however, a robust incorporation of **nondeterminism** into the learning framework.

Integration with Monitoring. In the CONNECT process, learned models of components should be monitored to detect evolutions of or previously unknown deviations from actual behavior. Therefore, the challenge is to devise and implement mechanisms to observe running CONNECTed systems, in order to provide feedback to the learning enablers in such a way that existing learned behavioral models can be continuously updated, based on actual observations taken in the field.

Tool support The tool support should be extended to support the conceptual advances made in CONNECT. Support should be included for data parameters, abstractions, and QoS properties. An easily deployable "learning studio" should be made available to project partners. Tool-integration with monitoring tools should be realized.

1.5 Overview of Achievements in Y2

This deliverable D4.2 describes our progress and achievements in Y2. In this section, we survey our achievements while providing an overview of the material in this deliverable. As background for the technical presentation in the subsequent chapters, we provide in Chapter 2 a summary of the basic technologies that make up the foundation for work in WP4. We first give a short overview of classical automata learning techniques, including the L^*_{M} learning algorithm, and also describe how a concrete learning setup can be constructed. A small example illustrates the process and the chapter concludes with a list of nontrivial issues that must be considered in a workable learning setup.

Algorithmic Improvements for Effective Learning Active learning relies on performing a significant number of membership queries (which can roughly be thought of as test cases) in order to generate a faithful model of a networked component. A major function of these queries is to search for yet undiscovered parts of the component's behavior. In some theoretical work on learning, the search for undiscovered behavior is encapsulated into complexity-hiding concepts, sometimes called "equivalence queries". Real learning must face the challenge to carry out this search efficiently. First, the number of employed queries has to be optimized significantly. Especially for equivalence approximation, which is theoretically considered to use a number of test runs that is exponential in the size of the model, a pragmatic approach has to be found. Second, and even more fundamentally, a feasible way of realizing (i.e., approximating) equivalence queries has to be found.

During Y2, we have made significant progress towards more efficient exploration of the behavior of a component. In Chapter 3, we present a learning algorithm that can be configured to have different characteristics wrt. resource consumption. Especially it can be used to save a considerable amount of membership queries compared with classic algorithms. We verified the power of our new approaches in the ZULU challenge [42], which we won, in competition with several very strong groups in the language learning community.

Learning Rich Automata Models with Data One of the main challenges for Y2 has been to extend learning techniques to handle rich models including data of various forms. During Y1, we have approached this problem by manually adding an *abstraction module*, which connects a rich model to a finite-state abstraction, which can be learned by classical techniques. To address the challenge of automating this activity, we must face the problem to make learning algorithms data-aware in a deeper sense. This means that the learning algorithm must be able to produce carefully selected membership queries that involve data, and to organize their results in a well-defined way. In the absence of manual guidance, the organization should be done according to principles for how to structure state machines extended with data. Unfortunately, there exist almost no good candidates in the research literature for how to "best" structure such state machines. For many classes of such machines, there are no known minimal forms, and so already here there is a challenge to overcome.

In Chapter 4, we present a solution to the above challenge, by developing a canonical form for automata with data, which significantly improves on previous work in this area. The result is a canonical form of *register automata*, i.e., finite-state machines extended with data variables, which is intuitive and succinct. The remarkable aspect of our work is that it allows to define a unique canonical automaton for any behavior, by means of generalizing the classical Myhill-Nerode theorem. In itself, this represents a significant contribution to the automata and modeling communities, since Myhill-Nerode theorems have been difficult to obtain for most extensions of finite automata. It also provides a very suitable basis for developing learning algorithms, since the Myhill-Nerode theorem allows most of the principles of finite-state automata learning to be carried over.

Based on this breakthrough, it is thereafter much more straightforward to develop learning algorithms for behaviors that combine control and data. In Chapter 5, we present an extension of the active learning algorithm L^* to generate rich component models. We have implemented our algorithm in the framework of the LearnLib tool [72], and applied it to a number of case studies.

Dynamic Refinement of abstractions During Y1, we have identified abstractions as an important concept in the process of learning behavioral models of realistic systems. We developed techniques for augmenting finite-state learning with abstractions in order to learn behavior of systems with data. In the previous paragraph, we described our progress during Y2 towards automating this process. One important part of such techniques is the ability to refine the abstraction when it is found to be inadequate. The typical sign of an inadequate abstraction is that it becomes non-deterministic. Intuitively, this can be understood as a manifestation of the fact that the abstraction blurs information in the input to the component so that the learning algorithm can no longer generate a causal link between input and output. The remedy in such a situation is to refine the abstraction so that the detrimental blurring is removed.

In Chapter 6 we introduce a method for refining a given abstraction to automatically regain a deterministic behavior on-the-fly during the learning process. Thus the control over abstraction becomes part

of the learning process, with the effect that detected non-determinism does not lead to failure, but to a dynamic alphabet abstraction refinement. Like automata learning itself, this method in general is neither sound nor complete, but it also enjoys similar convergence properties even for infinite systems as long as the concrete system itself behaves deterministically, as illustrated along a concrete example.

In the context of CONNECT this method will also be valuable when it comes to automatic generation of test-drivers (cf. Chapter 2). A major step in the automation of test-driver generation will be the automatic production of abstractions. It is highly unlikely that an approach can be found that always immediately generates a correct and meaningful abstraction. Automated alphabet abstraction refinement will enable the automatic correction of "good" abstractions (in a sense to be defined more precisely below) rather than trying to produce perfect abstractions. The remainder of this chapter is published as [60].

Effect-Directed Learning Within the context of the CONNECT project, automata learning is employed in ways that diverge considerably from, e.g., automata learning in the context of system validation. It thus makes sense to evaluate the requirements on automata learning techniques in a CONNECT-specific usage scenarios.

For instance, in a CONNECT scenario there may be a priori "relevant" information about scenarios. This includes the observance of preconditions for system actions, such as, e.g., the precondition that in an e-commerce system a user has to be logged in before any business transactions can be concluded. A model useful for synthesis can be restricted to those scenarios where any such preconditions are obeyed and, obviously, should also contain information on how to operate all relevant system features. Still, to ease the task of CONNECTor synthesis, it is desirable that the learned models only contain information on such system properties which are needed for proper operation of the CONNECTed system. This implies that a careful balance between (feature) completeness and model compactness has to be found.

In Chapter 7, we outline what the requirements are and propose a concept of "effect-centric" learning, which aims at decreasing time spent on learning while still guaranteeing a certain concept of "completeness", i.e., that the models contain relevant information.

Learning Non-functional Properties In the CONNECT framework, the learning enabler is supposed to gather information about non-functional properties while inferring functional models of networked systems. We are pursuing two lines of approach. One line is to consider latencies between interactions as a data parameter (e.g., in the form of time stamps), and use techniques like those mentioned under "Learning Rich Automata Models with Data" above to infer them. This line of work has progressed during Y2 with he work on [51].

Another line or work, is to augment an existing functional model with constraints on latencies between actions. In Chapter 8, we introduce the necessary extensions to learning technology alongside some results of a prototypical application of this new method. Considering the CONNECT architecture and data-flow (cf. Fig. 1.1), the learning enabler will get from the discovery enabler interface description of networked systems, ontology on data domain, and metrics of interest provided by networked system. While the first two will be used by the learning enabler to construct a test-driver and abstraction(s) as discussed in the previous chapters, the metrics of interest will be used to generate fitting probes into the learning process (see below). The learning enabler will produce *guarantees provided by the networked system* as (non-functional) output, which will be used during dependability analysis. Consider latency, i.e., response time as an example of such a guarantee: the learning enabler can provide the information that a networked system did react to some message within 20 ms in 95% of the cases and did always react within 200 ms. Further examples are throughput, failure rates or influence of load on the aforementioned metrics.

The CONNECT runtime monitoring infrastructure A challenge for Y2 has been to build and integrate an infrastructure to continuously monitor the runtime behavior of the CONNECTed systems to respond to the growing needs of evolution and adaptation. The events to be observed, belonging to guaranteed functional and non-functional properties, can themselves vary in scope and along time. To address this need, in D5.1 we had already discussed the requirements for such a monitoring infrastructure in CONNECT.

Such requirements are briefly summarized in Section 9. In particular, we expand here on the requirements of modularity and flexibility, which are key to support the integration with the various CONNECT enablers.

In Chapter 9, we describe the implemented infrastructure and show how it can be applied to parameterized models for various types of runtime analyses.

Algorithms for comparing automata, e.g., in equivalence queries The work on developing efficient algorithms for comparing hypothesis and target automata when performing equivalence queries in a white-box context [12], which was reported in D4.1 received the EATCS (European Association for Theoretical Computer Science) award for best theoretical paper at ETAPS 2010. The effort has been continued with the development of so far best algorithms for comparing automata over infinite behaviors [11].

1.6 Enhancement of the Tool Infrastructure

During Y2 many new concepts, as sketched above, were conceived, and subject to experimental evaluation. To enable this experimentation, our central tool, the LearnLib framework for automata learning, had to be extended accordingly. An overview on LearnLib is given in [84]. Work items on LearnLib included:

Implementation of test-drivers with data history: In many protocols values formerly transmitted play a key-role during communication. Thus a test-driver was created, which keeps a history of data values that can be used to construct learning queries. This functionality is important for automating the construction of abstraction modules, as in [8, 9]. The example presented in Section 2.3 employs this test driver.

Development of fast equivalence oracles: This was key to participating and and eventually winning the ZULU competition, giving valuable insight into concerns relevant to CONNECT scenarios. See Chapter 3 for an overview.

Initial support for rich models with data: During Y2 a rich, data aware model formalism was created, as described in Chapter 5. The implementability of this approach was verified using an early realization within the foundations of the LearnLib framework.

Implementation of automatic alphabet refinement: To obtain experimental results on our method of alphabet refinement (cf. Chapter 6), an implementation was created and tested on a scenario previously examined by Aarts et al. in [9].

Prototypical modules for effect-directed learning: In the context of CONNECT, it is important to create models fit for synthesis. Thus it is desirable to focus on relevant aspects of the system, while ensuring that all necessary information on the target systems is contained (cf. Chapter 7 and [58]). A proof-of-concept implementation was created.

Support for batches of learning queries: This enables parallel processing of queries, which allows to inflict high load on examined target systems. As a result non-functional data concerning the explored system can be collected in a more state approximating real-life load scenarios. More details can be found in Chapter 8.

Apart from extending the functionality of the LearnLib, work was also done on consolidation and documentation. A stable version of LearnLib was made available on `http://www.learnlib.de`.

2 Recap and Example

This chapter briefly recalls the basic concepts of active learning. The introduced notation will be be used in most remaining chapters. We also provide a small motivating example that shows the application of active learning in a CONNECT-specific case study.

2.1 The L^*_{M} Learning Algorithm

Query learning (or *active learning*) attempts to construct a deterministic finite representation, e.g., a Mealy machine, that matches the behavior of a given target system on the basis of observations of the target system and perhaps some further information on its internal structure. Here, we only summarize the basic aspects of our realization L^*_{M} for Mealy machines [79], which is based on Angluin's learning algorithm L^* for finite state acceptors [16]. A more elaborate version of this summary and an extended discussion of the practical aspects of active learning is given in [101].

Definition 1. *A Mealy machine is defined as a tuple* $Sys = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

- Q *is a finite nonempty set of* states *(let* $n = |Q|$ *be the size of* Sys*),*
- $q_0 \in Q$ *is the* initial state*,*
- Σ *is a finite* input alphabet*,*
- Ω *is a finite* output alphabet*,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*, and*
- $\lambda : Q \times \Sigma \to \Omega$ *is the* output function*.*

Intuitively, a Mealy machine evolves through states $q \in Q$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(q, a)$ and produces an output according to $\lambda(q, a)$[1]. We write $q \xrightarrow{i/o} q'$ to denote that on input symbol i the automaton moves from state q to state q' producing output symbol o.

Query learning is also referred to as *active* learning as it constructs Mealy machine models by actively querying the target system. It poses *membership queries* that test whether certain strings (potential runs) are contained in the target system's language (its set of runs), and *equivalence queries* that compare intermediately constructed hypothesis automata for equivalence with the target system. Learning terminates successfully as soon as an equivalence query signals success.

In its basic form, active learning starts with a hypothesis automaton with only one state and refines this automaton on the basis of query results iterating the three main steps shown in Fig. 2.1: *refining the hypothesis*, *conformance testing*, and *analyzing counterexamples*, until a state-minimal deterministic (hypothesis) automaton consistent with the target system is produced. Key to achieving this result is the Nerode-like dual characterization of states:

- by a set, $S \subset \Sigma^*$, of *access sequences*. This characterization of state is too fine, as different words $s_1, s_2 \in S$ may lead to the same state in the target system. L^*_{M} will construct such a set S, containing access sequences to all states of the target automaton. It will also maintain a second set, SA, which together with S will cover all transitions of the target system (SA will during the course of learning always be $SA = (S \cdot \Sigma) \setminus S$).
- by an ordered set, $D \subset \Sigma^*$, of *distinguishing sequences*. L^*_{M} realizes the characterization of hypothetical states q simply in terms of vectors $row(s) = \langle r_1, \ldots, r_k \rangle$ (with $r_i \in \Omega$), characterizing states by means of subsequent outputs: For $rows(s)$, let $r_i = \lambda(\delta(q_0, s), d_i)$.

[1] In the remainder of this deliverable, we will use an extended version of the transition function: $\delta' : Q \times \Sigma^* \to Q$ with $\delta'(q, aw) = \delta'(\delta(q, a), w))$, where $q, q' \in \Sigma$, symbols $a \in \Sigma$, and $w \in \Sigma^*$. The same holds for the output function.

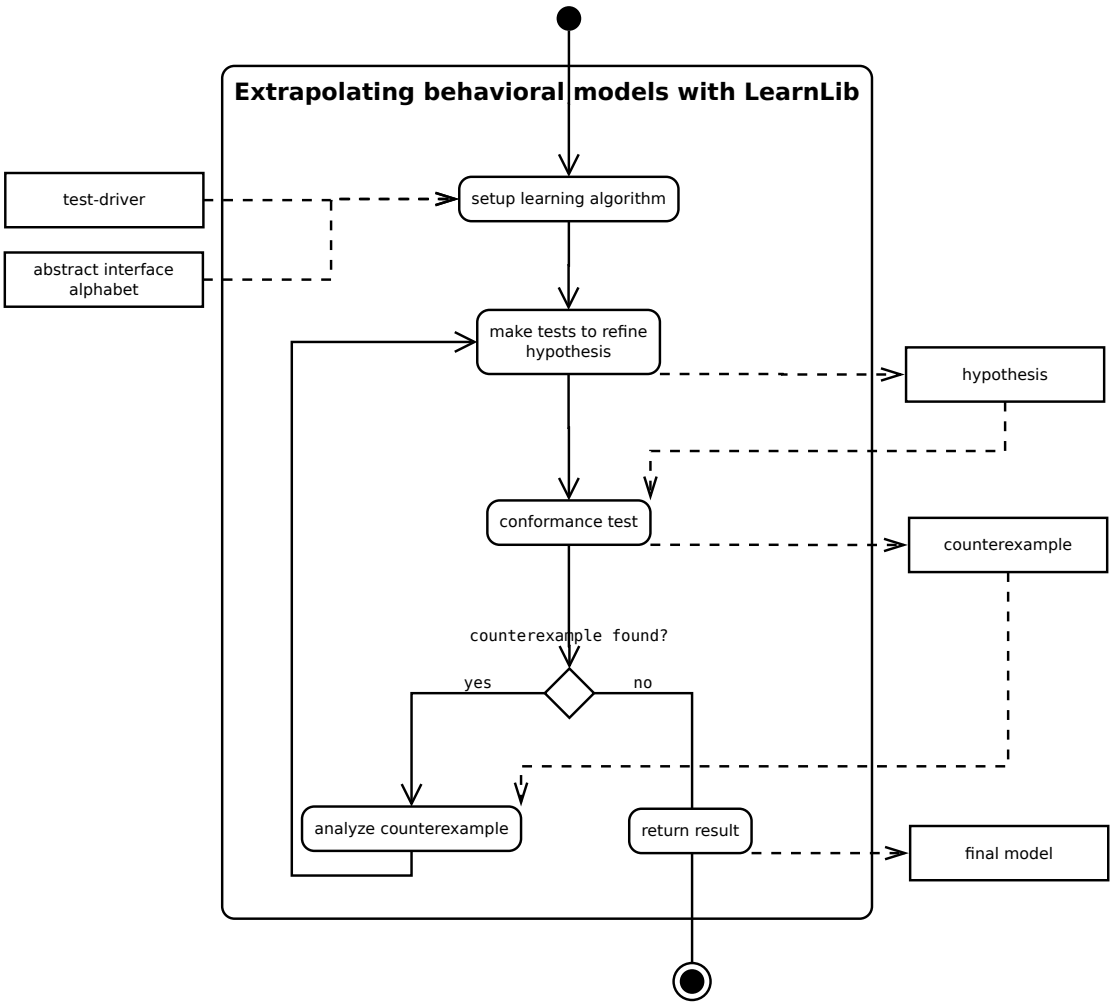

Figure 2.1: Structure of Active Learning Algorithms (modeled in XPDD [67])

The set S will be initialized as $\{\epsilon\}$, containing only the access sequence to the initial state; SA will accordingly be initialized as Σ, covering all transitions originating in the initial state. The ordered set D will be initialized as Σ, allowing to identify a state by the output that is produced along the transitions originating in this state. The learning procedure then proceeds as shown in Fig. 2.1 by:

Refining the Hypothesis: This first step again iterates two phases. The first phase checks whether the constructed automaton is closed under the one-step transitions, i.e., each transition from each state of the hypothesis automaton ends in a well defined state of this very automaton. This is the case if for every $r \in S \cdot \Sigma$ there exists a $s \in S$ with $row(s) = row(r)$. Otherwise, S will be extended by the corresponding r until *closedness* is established (and SA will be extended accordingly). This extension is guaranteed to result in a unique fixpoint, independent of the order in which the rows are processed.

The second phase then checks whether two access sequences $s_1, s_2 \in S$ with the same bit vector, $row(s_1) = row(s_2)$, have also the same outgoing transitions, a necessary precondition for them to represent the same state. This condition, which is called *consistency*, can be formalized as follows:

$$\forall s_1, s_2 \in S \; \forall a \in \Sigma \; . \; row(s_1) = row(s_2) \stackrel{?}{\Rightarrow} row(s_1 \cdot a) = row(s_2 \cdot a).$$

It is easy to see that detected inconsistencies can be removed by elaborating the set D of distinguishing futures in a way that makes some of the difference observed on a distinguishing transition visible before that transition: one simply needs to prefix the distinguishing future that separates the two target states on the distinguishing transition by the label of this very transition.

Unfortunately, such additions to D may break the previously achieved completeness, which requires to re-iterate the completion procedure. This, in turn, may lead to new violations of consistency. However, successive iteration of these two steps is guaranteed to result in a unique, well-defined, closed, and complete hypothesis automaton whose states are characterized by the bit vectors. In more detail:

- every state $q \in Q$ of the hypothesis automaton is reachable by at least one word $s \in S$, i.e., $row(s)$ corresponds to q,
- there exists a transition $\delta(q, a) = q'$ *iff* there exists $s \in S$ with s reaching q (or with $row(s)$ corresponding to q) and $row(s \cdot a)$ corresponding to q',
- The output function can be constructed from the $row()$-vectors as well. As D is initialized as $Sigma$, the values for all $\lambda(q, a)$, where $a \in \Sigma$, are contained in the $row(s)$ vector corresponding to q.

Conformance Testing & Counterexamples: After closedness and consistency have been established, an equivalence query checks whether the language of the hypothesis automaton coincides with the language of the target automaton. If this is true, the learning procedure successfully terminates. Otherwise, the conformance test returns a counterexample, i.e., a word which distinguishes the hypothesis from the target automaton. All prefixes of a counterexample will be added to S (SA will be extended accordingly). This will lead to inconsistency, which in turn will lead to a new distinguishing suffix [79].

The correctness argument for the whole approach follows a straightforward pattern.

1. The state construction guarantees that the number of states of the hypothesis automaton can never exceed the number of states of the smallest (minimal) automaton, behaviorally equivalent to the target system.
2. The treatment of counterexamples guarantees that at least one additional state is added to the hypothesis automaton each round. Thus, due to 1), such treatments can happen only finitely often.
3. The conformance testing (or equivalence query) provides new counterexamples as long as the language of the hypothesis automaton does not match the desired result.

Put together, this guarantees that the learning procedure terminates after at most n rounds with a minimal automaton, behaviorally equivalent to the target system, where n is the number of states of this automaton.

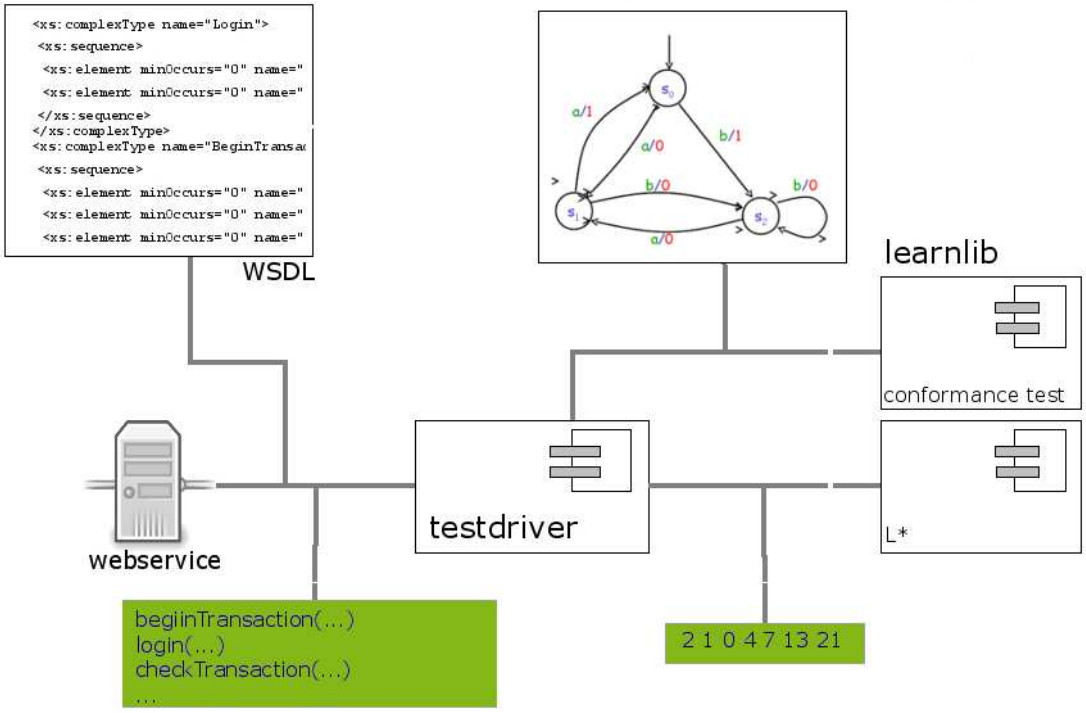

Figure 2.2: Experimental learning setup

2.2 Experimental Setup

One can identify a simple pattern in all practical scenarios: a learning algorithm is instantiated with an (abstract) set of input symbols for the system to be inferred. The membership queries (sequences of input symbols) the algorithm produces in the course of learning, are then passed to a special test driver. The test driver operates the system to be inferred by 1) translating the membership queries into sequences of actions on the real system and 2) performing these actions one-by-one on the system. For each performed action a third component, a logging facility, records the reactions of the actual system. After the execution of a sequence of inputs has finished, the resulting sequence of reactions is translated into a sequence of (abstract) symbols understood by the learning algorithm and passed back (to the learning algorithm).

This leads to an experimental setup as shown in Fig. 2.2. A learning algorithm (e.g. from LearnLib) is connected to a special runtime environment (in this case, e.g., containing means to communicate with a web service). The runtime environment instantiates a connection to the system under test and exposes some test driver. The test driver then executes actions on the system under test on behalf of LearnLib, while the virtual machine (or some other adequate facility) monitors the system reaction. The reactions are translated by the test driver and the translated versions are reported to LearnLib.

Active automata learning is characterized by its specific means of observation, i.e., its proactive way of posing membership queries and equivalence queries. Thus it requires some effort to realize this query-based interaction for the considered application contexts. The general requirements are shown in Fig. 2.3.

As shown in Fig. 2.3, a test-driver can be generated from an *interface description* of the system under test (SUT), an *instance pool*, and a *reset strategy*. Sometimes, by considering a *learning purpose*, uninteresting parts of the SUT can be hidden from the learning algorithm. The interface description will be used to generate code to *instrument the SUT*. The instances that shall be assumed for data values in the communication with the SUT and the purpose allow the formulation of a mapping between abstract interface symbols and concrete actions on the SUT. Finally, the reset strategy will ensure that all tests on the SUT will be executed under the same initial conditions.

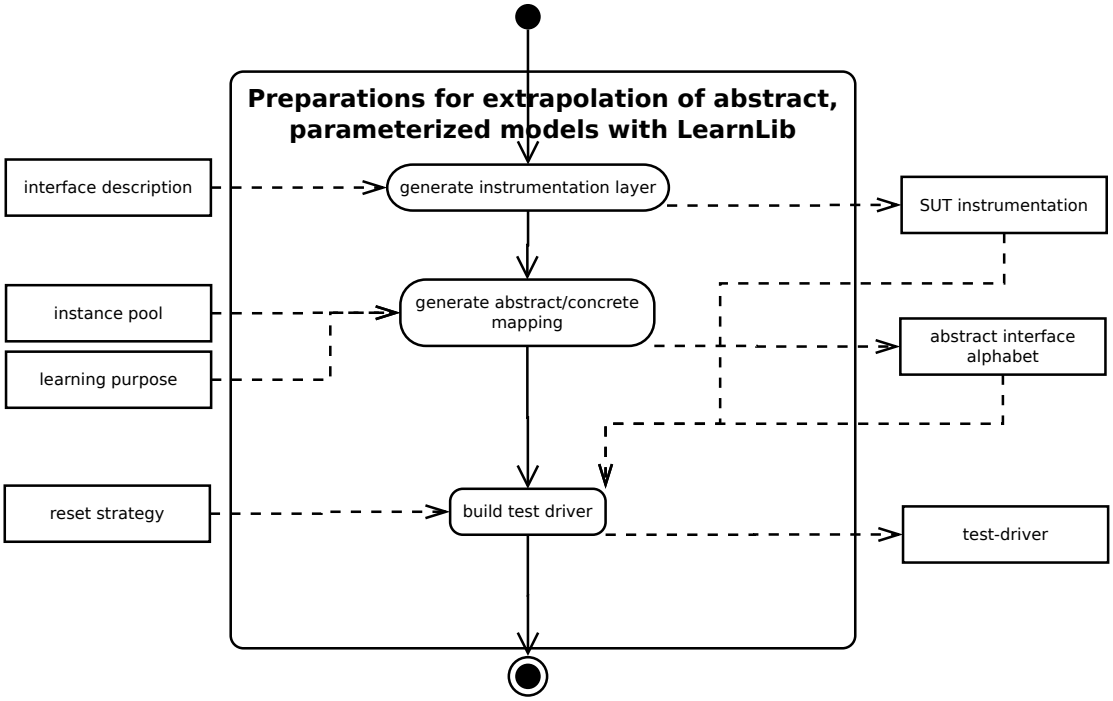

Figure 2.3: Test-driver & Alphabet Generation (modeled in XPDD [67])

2.3 An Introductory Example

Let us introduce a small example to illustrate the discussed ideas. Imagine a system for booking seats in events. A user of this system has to provide his credentials and can then browse through a list of venues. For each venue a list of available seats can be accessed. Finally, from these lists of seats a single seat can be booked, which will be confirmed in a corresponding receipt.

Table 2.1: Interface Descriptions

Interface 1	Interface 2
-	session ← openSession(user,pwd)
venue[] ← getVenues(user,pwd)	venue[] ← getVenues(session)
seat[] ← getSeats(user,pwd,venue)	seat[] ← getSeats(session,venue)
receipt ← bookSeat(user,pwd,seat)	receipt ← bookSeat(session,venue,seat)

For this system we will define two slightly different interfaces. The signatures of the interfaces' methods are given in Table 2.1. The differences between the two interfaces are:

1. In *Interface 1* credentials have to be provided in all invocations, while in *Interface 2* the credentials are only used once to create a session identifier. This identifier will then be used in subsequent invocations.
2. Assuming that the methods of the interfaces will be called from top to bottom in order to actually book a seat, calling the first method of *Interface 1* corresponds to calling the first two methods of *Interface 2*.
3. While in *Interface 1* to book a seat, only the identifier for this seat has to be provided, in *Interface 2* the identifier of the corresponding venue has to be provided as well.

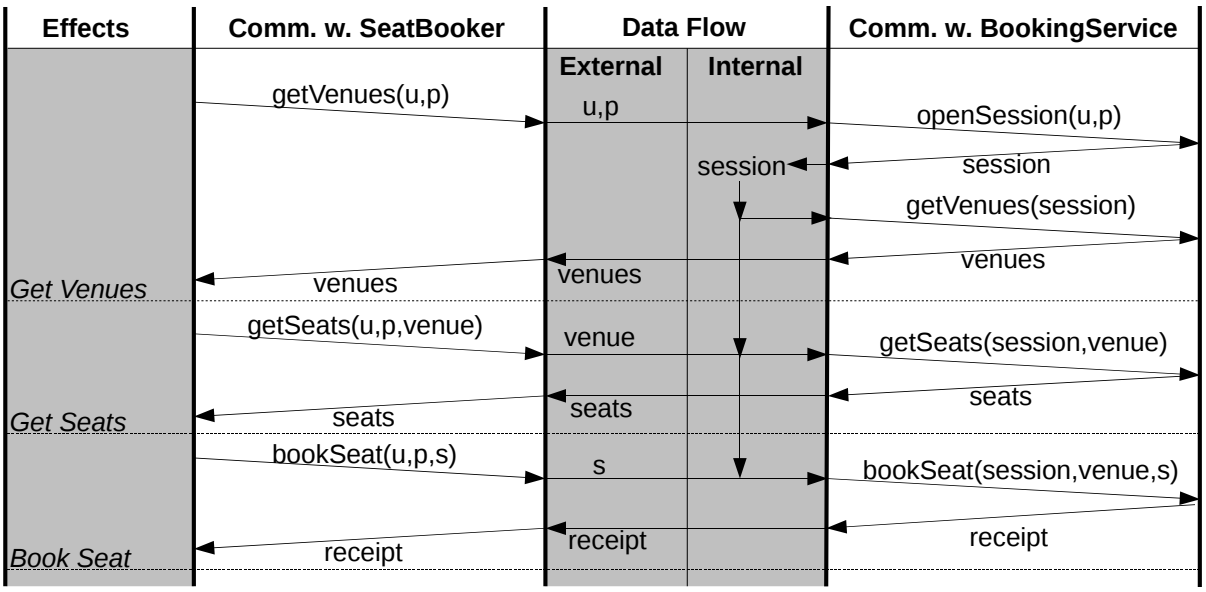

Figure 2.4: Message Sequence Chart for Connected Implementations

We now want to connect a front-end of the system, which uses *Interface 1*, to a back-end using *Interface 2*. While the actual construction of a CONNECTor for the two parties is out of the scope of this deliverable, we will consider here the general requirements on the inferred models of the two networked components that can be derived from this example. Naturally, there will arise a number of other problems when generating a CONNECTor, like transforming data parameters from one format into another, or actually instrumenting the components, etc. However, these problems do not directly lead to requirements on the obtained models.

In the message sequence chart in Figure 2.4, it is shown how the two implementations can be connected. The first column of the figure shows the synchronization of the two implementations on the shared effects. The second column shows the communication between the component using *Interface 1* and the CONNECTor, while the fourth column shows the communication between the CONNECTor and the second implementation using *Interface 1*. In the third column, the data-flow between the two implementations is visualized. Here, it is interesting to observe that there exist parameters which do not exist in both interfaces (e.g., session). We refer to those parameters as *control*-parameters as they are only used to control one party and need not to be passed to the other party. The model that is to be constructed by regular inference is now expected to expose exactly this kind of causal relations between the formal parameters of the different primitives (that is has to be same session identifier in all the calls).

Learning the seat booking service Imagine a web service realizing the seat booking system and using the interface from the right half of Table 2.1. For the interface description WSDL-S [13] is used, which allows

1. annotating primitives with effects,
2. annotating parameters with (abstract) data types.

Technically, this is realized by references into an ontology. A common ontology can then be used to link several heterogeneous systems with the same intention on a semantic level.

We generated Java proxy-classes for the web service. Using standard Java APIs, the proxy classes were inspected for methods, which are used as learning alphabet. Also, the WSDL-S document was analyzed for data types and effects.

Using the information about data types, a parameter structure was generated: The return values of method invocations were stored in the local parameter structure of the test-driver for usage in future invocations of the target system. The type and name of the output symbol determined the parameter, a returned

Table 2.2: Parameter Structure and Effects

Symbol	Output-Parameters	Store as	Effect
openSession(user,pwd)	Session Identifier	session	-
getVenues(session)	List of Venues	venues	GetVenues
getSeats(session,venue)	List of Seats	seats	GetSeats
bookSeat(session,venue,seat)	Receipt	receipt	BookSeat

(output-)value will be stored in. The resulting parameter structure is shown in Table 2.2. Input-values for method invocations were either

- predetermined values, e.g., for `user` and `pass` in the `openSession` primitive,
- references to parameters, e.g, for `session` in all primitives, or
- simple expressions over parameters, e.g., `!session` (not session) and `(in)venues` (denoting a venue from a list of venues).

The *abstract interface alphabet* was generated over all possible combinations of data and primitives. The resulting abstract input alphabet is shown in Table 2.3. The corresponding test-driver (cf. Section 2.4) invokes methods on the WSDL proxy according to the alphabet symbol currently processed.

Table 2.3: Abstract Input Alphabet

Interface Symbol	Input-Parameters
openSession(u,p)	u='test',p='test'
getVenues(si)	si=session
	si=!session
getSeats(si,v)	si=session, v=(in)venues
	...
	si=!session, v=(!in)venues
bookSeat(si,v,s)	si=session, v=(in)venues, s=(in)seats
	...
	si=!session, v=(!in)venues, s=(!in)seats

All data values used in this examples were sequences of characters. The `!`-function was realized by adding some characters to the corresponding sequence. The `(in)`-function was realized by picking the first element of a list, which would also ensure that two subsequent applications of `(in)` return the same value. The `(!in)`-function was realized by constructing a value, which was not contained in the corresponding list.

To distinguish (i.e., classify) states in active learning, some kind of output is needed from the system under test. As the returned values in this example are stored in the parameter structure and thus are treated symbolically, we used the WSDL 'fault' returns as classifiers: In WSDL, every interface method can return with a normal or with a fault answer. In Java, this fault case corresponds to an exception being thrown. The test-driver would simply catch and evaluate these exceptions. In the example, three effects were annotated to the normal return of the interface methods (cf. Table 2.2).

Since we treat the return values symbolically, an error message that is encoded within a normal answer will, without further analysis, not be treated as a fault. Thus, if the SUT does not use use fault answers for error signaling, resorting to encapsulating error messages in regular messages, an introspection of received messages is necessary to determine error cases. An approach to detect errors by inspecting returned messages was implemented in the StrawBerry tool, presented in [26]. This tool, developed by CONNECT project partners, is slated for integration with the learning tools during Y3.

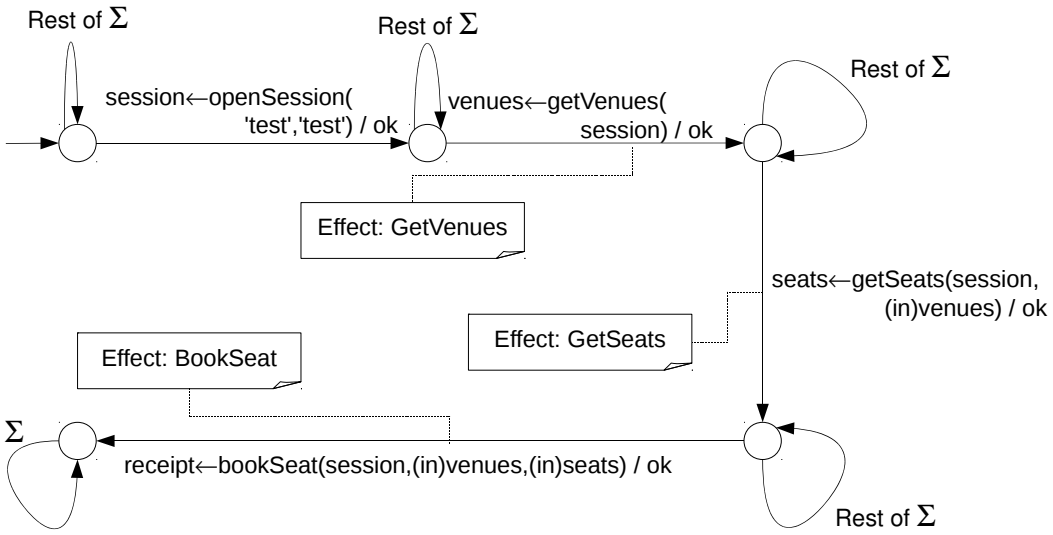

Figure 2.5: Behavioral Model of the Seat Booking System

Results and conclusion for this example The behavior of the web service was learned using LearnLib [92], the automata learning framework developed at TU Dortmund. The resulting model with 5 states is shown in Figure 2.5. This model clearly shows what steps are necessary to book a seat from start to finish and also shows what data values are to memorized and where to use them later on. With this information, it is easy to derive a client that accesses the provided service and mediates successful transactions.

This example demonstrates how automata learning can be employed to learn a real system. For this example only a very small system was learned, yet it is demonstrated how concerns like interfacing to the system, abstraction, alphabet construction, and data-handling come into play.

2.4 Automated Test-driver Generation

For the presented example, the test-driver was put together manually from

- automatically generated instrumentation code,
- manually constructed and analyzed information from the WSDL-S description,
- manually provided fixed values,
- manually implemented functions on data parameters.

To fully automate the generation of adequate test-drivers, the single steps will subsequently have to be automated (where possible). Implementing solutions will be tightly coupled to the integration work that is a key focus for Y3, as relevant technology is either preexisting or developed by project partners within their work packages:

Generation of instrumentation code: For services employing standardized remote invocation techniques, there usually exist tools that create networked proxies that expose functionality as described in interface description. E.g., for WSDL services and the Java platform, there does exist the standard tool `wsimport`, which creates proxy classes that can be directly called to invoke methods on the remote system. Similar tools are available for other remote invocation schemes, such as CORBA. Thus the task to generate instrumentation code can be delegated to a library of tools that generate instrumentation code from interface descriptions.

Determining relations between methods and data values: For meaningful interaction with the target system, most remote method invocations have to include data values, as demonstrated in the example above. To determine what remote methods can potentially produce data valuable for other method invocations, the system's interface description has to be analyzed. The generated information of this analysis is of direct use for creating the parameterized alphabet of the learning process. In [26] an approach and tool for the analysis of WSDL descriptions is presented, which will be integrated during Y3.

Providing fixed data values: Some predetermined data values, such as, e.g., credentials, are not communicated during normal interaction with remote systems. These values, however, are often necessary to access crucial parts of the target system, and should either be included in the interface description or determined by, e.g., the discovery enabler in the CONNECT architecture (cf. [21]).

3 Improving Classic Algorithms

One purpose of the CONNECT architecture is the (online) generation of CONNECTors at runtime: the CONNECT enablers will detect when a networked system tries to interact with another system. At some point in the process of producing a fitting CONNECTor, the learning enabler will learn a model of both networked systems. As discussed in the introduction, learning usually is an expensive enterprise (in terms of tests that have to be executed in order to produce a model).

The costs in learning have two sources: (1) the membership queries (test runs) used in the learning process and (2) the test runs used to approximate equivalence queries (equivalence cannot be computed with a finite amount of tests). Additionally, in practice even single test runs can be quite expensive. For these reasons unoptimized learning algorithms will fail to produce models for real systems with more than a couple of states and interface actions in a feasible amount of time.

To enable the online generation of CONNECTors in practice, learning has to be optimized in two respects. First, the consumption of test runs (both in terms of membership queries and equivalence queries) has to be reduced drastically. Especially for equivalence approximation, which is usually considered to use a number of test runs that is exponential in the size of the model, a solution has to be found.

Second, and even more fundamentally, a feasible way of realizing (i.e., approximating) equivalence queries has to be found. In simulation scenarios conformance testing methods are often used to approximate equivalence queries. These, however, rely on additional assumptions on the system under test (SUT), which in general cannot be made, e.g., a fixed upper bound on the size of the SUT (in terms of the number of states) is assumed. The additional assumptions are used as termination criteria. They determine when a sufficient coverage with tests is reached.

In this chapter, we present a learning algorithm that can be configured to have different characteristics wrt. resource consumption; indeed, it can be used to save a considerable amount of membership queries compared with classic algorithms. Furthermore, we present a novel approach to equivalence approximation: It has turned out that changing the view from 'trying to prove equivalence', e.g., by using conformance testing techniques, to 'finding counterexamples fast' has a strong positive impact.

We verified the quality of the new approaches in the ZULU challenge [42], which we won. Thus, the remainder of this chapter, which has been published as part of [59], discusses the improvements mainly in the context of ZULU. The improvements, however, will have a significant impact in the CONNECT context, too (e.g., the patterns presented here will be the essential basis of the framework for learning non-functional properties that will be introduced in Chapter 8).

The chapter also summarizes our experience with the ZULU challenge on active learning without equivalence queries, presents our winning solution, and

- investigates the character of ZULU's rating approach, which ranks solutions according to their prediction quality for language containment based on a ZULU-generated test suite, and
- discusses how this approach can be taken further to establish a framework for the systematic investigation of domain-specific, scalable learning solutions for practically relevant application.

Especially the second topic is relevant in the context of CONNECT.

3.1 The ZULU Competition

ZULU [41] is a competition in active learning from membership queries: contestants had to develop active learning algorithms (following the general approach due to Dana Angluin [16]). Essential to ZULU have been two main ideas:

No equivalence queries: Equivalence queries (in the theoretical formulation of active learning) compare a learned hypothesis model with the target system for language equivalence and, in case of failure, return a counterexample exposing a difference. Their realization is rather simple in simulation scenarios: if the target system is a model, equivalence can be tested explicitly. In practice, however,

the system under test will typically be some kind of black-box and equivalence queries will have to be simulated using membership queries [62]. One goal of the ZULU competition was to intensify research in this practical direction.

Limited membership queries: Equivalence queries can be simulated using model-based testing methods [33] (The close relation between learning and conformance testing is discussed in [22]). If, e.g., an upper bound is known on the number of states the target system can have, the W-method [38] or the Wp-method [47] can be applied. Both methods have an exponential complexity (in the size of the target system and measured in the number of membership queries needed).

By allowing only a (very) limited number of membership queries for every problem, the ZULU competition forced the contestants to change the objective from 'trying to prove equivalence', e.g., by using conformance testing techniques, to 'finding counterexamples fast'.

These two ideas led to the following concrete competition scenario. The contestants had to compete in producing models of finite state acceptors by means of membership queries, which the ZULU platform would answer. For every task, the number of granted membership queries was determined by the ZULU system as the number of membership queries needed by some basic Angluin-style reference implementation to achieve a prediction quality of over 70%. Solutions were ranked according to their quality of prediction relative to a ZULU-generated suite of 1800 test words (the same suite as used to determine the number of allowed membership queries).

During the competition, all contestants had to compete in 12 categories: 9 'regular' categories ranging over finite state acceptors with (a) growing numbers of states and (b) growing alphabets, as well as 3 'extra' categories, in which the number of granted membership queries was reduced further. The overall winner was determined as the contestant with the best average ranking. A detailed description of the technical details can be found on the ZULU web page [42].

The background of our team at the TU Dortmund is the development of realistic reliable systems [91, 92]. Accordingly, we are working on techniques to make active learning applicable to industrial size systems. Of course this also requires (1) saving membership queries and (2) finding counterexamples fast, but for man-made systems. It turned out that dealing with randomly generated systems changes the situation quite a bit, as systems generated randomly tend to have uniform topological structure. Man-made systems, in contrast, often show distinctive structures which can be taken advantage of by means of specialized exploration strategies. Our solutions therefore arose in a two step fashion:

1. a generically optimized basis, not exploiting any knowledge about the structure or origin of the considered target systems. The main strategy here was to follow an *evolutionary* approach to hypothesis construction which explicitly exploits that the series of hypotheses are successive refinements. This helps organizing the membership and equivalence queries.
2. a subsequent customization for exploiting the fact that we are dealing with randomly generated systems. Here we were able to exchange our traditionally breadth-first-oriented approximations of an equivalence oracle, which exploit ideas from conformance testing, by a very fast counterexample finder. Also this customization benefits from the evolutionary character of the hypothesis construction.

The careful experimental investigation of the above options and its variants profited from our learning framework, the LearnLib [92], which we recently considerably extended. In particular, it enabled us to build a highly configurable learning algorithm, which can mimic most of the known algorithms, as well as all our variations in a simple fashion. In addition, the experimentation facilities of LearnLib, comprising, e.g.,remote execution, monitoring, and statistics, saved us a lot of time. We are planning to provide contestants of the RERS challenge [36] with all these facilities in order to allow contestants a jump start.

3.2 A Configurable Inference Framework

The main algorithmic pattern we used for the ZULU competition can best be described as generalized *Observation Pack* [17]: a pack is a set of components (which usually form the observation table). In

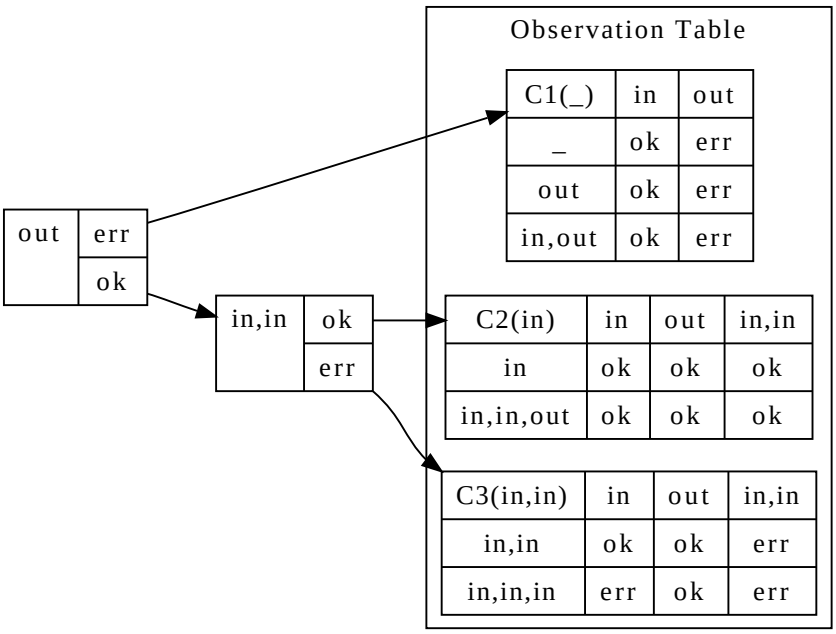

Figure 3.1: Extended Observation Packs

the original discussion an observation pack is introduced only as a unifying formalism for the algorithms of [16, 70]. We used a combination of discrimination trees [70] and reduced observation tables [93] to actually implement an observation pack. Fig. 3.1 illustrates the resulting data structure for a Mealy machine example (a three element buffer).

The observation table, which is a central data structure for most algorithms based on L^*_M as presented in Section 2.1, is split into independent components (essentially small observation tables in their own), each representing one state of the current hypothesis model, and being defined by one access sequence from the set S (see upper left corner). The second row of each component corresponds to the usual characterizing row for this access sequence in terms of some distinguishing futures taken from D, which label the columns. Additional rows stand for words of SA which happen to possess the same characterization in terms of the considered distinguishing futures, and therefore represent the same state in the hypothesis model. Please note that the description above does not hold for the third component which still needs to be split according to its difference with respect to the distinguishing future in (see below).

The left part of Fig. 3.1 shows the corresponding discrimination tree. Its nodes are labeled with elements of D, and its edges with the corresponding outcome of a membership query (here `ok` and `err` - one could also have chosen accept, non-accept). In the considered situation the discrimination tree simply reflects the fact that the first component can be separated from the other two due to the outcome concerning the distinguishing future `out`, whereas the second and third component can be separated from each other due to the outcome concerning the distinguishing future `in,in`. As indicated already, the third component could be split further due to the outcome concerning the distinguishing future `in`. Such a situation can only arise in the Mealy case, where the set of distinguishing futures is initialized with the full alphabet. In this concrete case, the alphabet symbol `in` is contained in the initial set of distinguishing futures but not yet in the discrimination tree.

In order to enter a new access string s into the discrimination tree, one starts with the root of the discrimination tree and successively poses the membership queries $s \cdot f$, where f is the distinguishing future of the currently considered node, and continues the path according to the corresponding output. If a corresponding continuation is missing, the discrimination tree is extended by establishing a new component for the considered access string. Finally, splitting a component requires the introduction of a new node in

the discrimination tree, like e.g. for the distinguishing future `in` to split the third component.

This realization allows us to easily switch between different strategies for handling counterexamples (e.g., [93, 76, 98]), as well as to use non-uniform observation tables, i.e., observation tables where for different access sequences different distinguishing futures may be considered.

To enable working with non-uniform observation tables, we extended the strategy for analyzing counterexamples from [93]. In its original form, this strategy produces a new distinguishing future d by means of a binary search over the counterexamples. During the search prefixes of the counterexamples are replaced by access sequences to the according states in the hypothesis model, in order to maintain as much of the hypothesis model as possible. Taking this idea a bit further allows us to guarantee that the access sequence s for a new component is always taken from SA. This automatically maintains the structure of the spanning tree, and it guarantees that only the component containing s is refined by the new distinguishing future d. The result of this refinement is a new node in the discrimination tree and two components (following the approach from [70]).

For ZULU, we configured two versions of our learning algorithm, both using this new strategy for analyzing counterexamples. The registered algorithms differed as follows.

Initial set of distinguishing futures: In one configuration, the initial set of distinguishing futures was initialized as $\{\epsilon\}$ (as in the literature). In the other configuration, we used $\{\epsilon\} \cup \Sigma$ in order to simulate the effect of changing from DFA to Mealy models. Please note that considering the empty word also for the Mealy configuration is a technical trick to better deal with the DFA-like systems considered here.

Observation Table: We used uniform and non-uniform observation tables. In a non-uniform table, the sets of distinguishing futures are managed independently for each component (cf. [17]).

Both decisions emphasize the same trade-off. Using the complete alphabet in the initial set and using a uniform observation table can be understood as a heuristic for finding counterexamples (in form of *unclosure*) in the table and thus reducing the number of equivalence queries. Using a non-uniform table and only proven distinguishing futures leads to using less membership queries but more equivalence queries.

3.3 Continuous Equivalence Queries

Essentially, active learning algorithms proceed in rounds triggered by negative outcomes of *equivalence queries* for successively improved hypotheses: returned counterexamples are analyzed and exploited to initialize the next round of refinement. Classically, each of these rounds were considered independent, and in fact, equivalence queries were simply provided with the current hypothesis model, which itself was independently constructed for each round. No knowledge about the algorithmic history was exploited. This is not too surprising for two reasons:

- Classically, equivalence queries were considered as atomic, with no need for a dedicated optimization, a point of view also supported by experimental settings, where the target systems are given as automata, which can efficiently be compared with the hypothesis automata.
- However, there is also a more technical argument. The hypothesis automata, which are refined during the learning process, are defined solely by the state characterization from above (their characterizing set), which is not fully reflected in the hypotheses. Thus the knowledge of the hypotheses alone is insufficient to establish the required notion of refinement.

In the meantime it is widely accepted that membership query-based approximations are key to practical application. The ZULU challenge itself is a very strong indication of this new perspective. Moreover, there are ways to strongly link the characterizations from below and above in a way that allow some kind of *incremental hypothesis construction* by means of a global, continuous equivalence querying process. Key

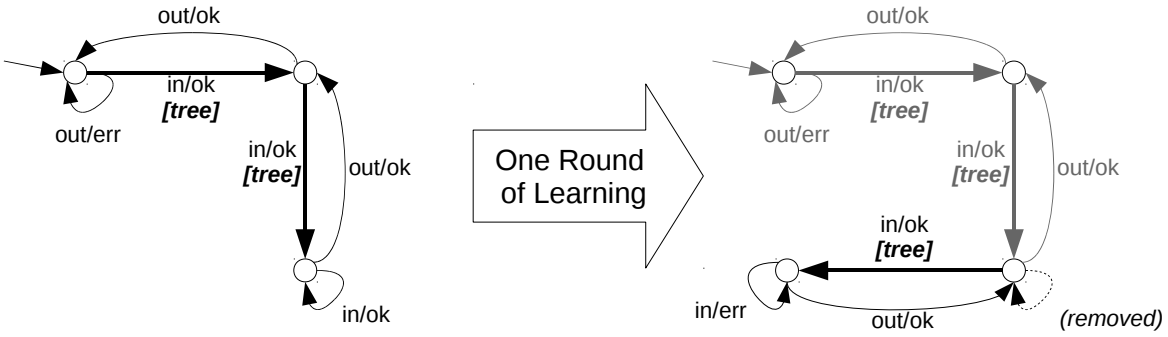

Figure 3.2: Evolving Hypothesis

to this approach are Rivest's and Schapire's reduced observation tables together with their way of analyzing counterexamples [93]). In their setting, the prefix-closed set S can be understood as a successively produced spanning tree of the target automaton, whose set of nodes is extended by elements from the SA-set. Thus there is always a unique representation of the states of the hypothesis automaton in terms of the monotonically increasing, prefix-closed set S. This invariant is maintained by the specific treatment of counterexamples:

> Each counterexample is minimized by means of binary search to a word/string sa of SA, followed by a distinguishing future d that forces sa to end up in a new state.

This form of counterexample allows maintaining the above invariant by moving the corresponding s from SA to S, add d to the set of futures D, and to continue with the usual procedure establishing closedness. Besides avoiding the construction of hypotheses from scratch, this procedure leads to a sequence of incrementally refined hypotheses, which allows for organizing the equivalence tests from a global perspective, sensitive to all the tests which have been performed in earlier iterations.

Fig. 3.2 shows how an evolving hypothesis is refined during one round of the learning process. The hypothesis in the left of the figure corresponds to the extended observation packs from Fig. 3.1. All transitions are labeled by annotations, indicating whether they belong to the spanning-tree or not. This information is vital, as it indicates proven knowledge. Adding a new state to the hypothesis leaves most of the original hypothesis and its annotations unaffected. In the example of Fig. 3.2 only the former loop at the third state is modified and two new transitions are introduced.

Beyond being a possible target for counterexample reduction (after some oracle provided an arbitrarily structured counterexample), the counterexample pattern

$$c = sa \cdot d, \text{ with } sa \in SA \text{ and } d \in \Sigma^+.$$

turned out to be ideal for realizing approximations of equivalence oracles using membership queries, or better, for implementing a method for revealing counterexamples fast along the lines shown in Fig. 3.3. Counterexample candidates sa are are tested for some heuristically chosen futures d until either a counterexample is found or some termination criterion is met. The effectiveness of the search heuristics for selecting counterexample candidates and finding distinguishing futures d may strongly depend on side knowledge. For the ZULU challenge, we exploited our knowledge that the languages were randomly generated as follows:

Select transitions & Book keeping: For the *E.H.Blocking* algorithm, transitions from the SA-set were chosen randomly. Once used, a transition was excluded from subsequent tests. When no transitions were left to choose from, all transitions were re-enabled. The *E.H.Weighted* algorithm uses weights

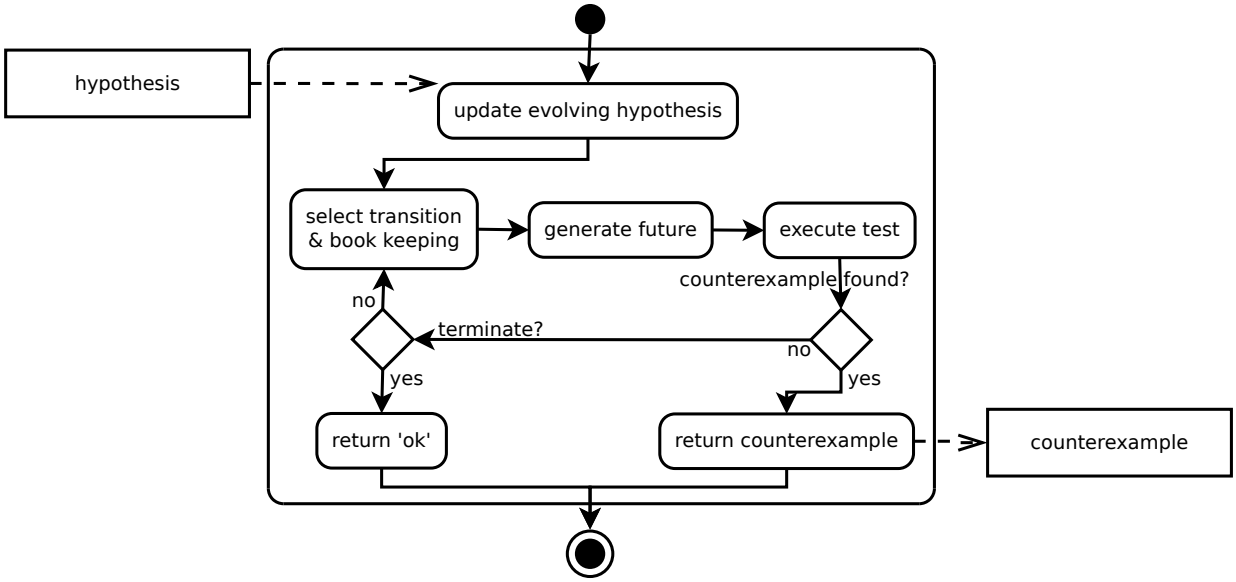

Figure 3.3: Continuous Equivalence Query

on all transitions, which are increased each time a transition is selected, and chooses transitions with a probability inversely proportional to its weight.

Generate futures: The futures were generated randomly with increasing length. The length was initialized as some ratio of the number of states of the hypothesis automaton, and was increased after a certain number of unsuccessful tests. The exact adjustment of the future length was developed in a experimentally to fit the properties of the problems in the ZULU challenge. This strategy of guessing comparatively long futures d turned out to be rather effective. It reduced the number of required membership queries to an average of 2-4, which radically outperformed our initial breath-first trials. Of course, this is very much due to the profile of the ZULU examples, and indeed, this choice of futures was the only truly domain-dependent optimization we developed.

We did not use an explicit termination criterion. A query terminated as soon as the number of queries granted by ZULU was exhausted.

3.4 Results

For the actual competition, we registered six candidate algorithms, split in two groups of three:

- the first group used a non-uniform observation table with a DFA-style initial set of distinguishing futures, and
- the second group used a uniform observation table with a (modified) Mealy-style initial set of distinguishing futures.

Both groups were equipped with the same three equivalence checking algorithms: (1) E.H.Blocking, (2) E.H.Weighted, and (3) plain random walks. As the random walks algorithm simply tested randomly generated words, the option for continuous equivalence queries did not apply. Table 3.1 shows the configuration of the algorithms, their average scores during the training phase and the eventual rankings from the competition.

Apparently, group 1 is far superior: about 10 points, and this in a setting where there are only 0.45 points between the first and the sixth place. In order to understand these results better, we investigated Problem

Table 3.1: Algorithms: Configuration and Ranking

Algorithm	Dist. Set		Equivalence Query		Training	Rank
	Init.	Uniform	Continuous	Strategy	(Avg.)	
E.H.Blocking	$\{\epsilon\}$	no	yes	block transitions	89.38	1
E.H.Weighted			yes	weight transitions	89.26	2
Random			no	random walks	88.93	6
run_random	$\{\epsilon\} \cup \Sigma$	yes	no	random walks	80.17	14
run_blocking1			yes	block transitions	79.89	15
run_weighted1			yes	weight transitions	79.65	16

Table 3.2: Detailed Training Example: Problem 49763507

Algorithm	New Membership Queries			Rounds	States	Score
	Close Obs.	Analyze CEs	Search CEs			
E.H.Blocking	6,744	358	999	259	352	94.11
E.H.Weighted	6,717	349	1,035	262	351	94.61
Random	6,586	519	996	228	332	93.28
run_random	8,080	14	7	5	312	74.89
run_blocking1	8,074	11	16	6	319	73.06
run_weighted1	8,077	9	15	6	319	74.39

49763507 in more detail. In particular we wanted to understand how the ZULU ranking mechanism, which is based on predictions rates for randomly generated test suites, reflects the quality of Angluin-style implementations of automata learning.

3.5 Discussion of the ZULU Rating Approach

In order to better understand the progress of the learning algorithms, let us consider some profile data of the training problem 49763507 in more detail, for which the ZULU environment allowed 8101 membership queries. Table 3.2 classifies the consumption of these 8101 queries according to their usage during (1) the closure of the Observation Table, (2) the analysis of counterexamples, and (3) the search for counterexamples. Moreover, it shows the number of learning rounds performed, the detected states, and the eventual scores.

These results are quite drastic:

1. The difference between the two groups discussed in the previous section is even more impressive here. It is obvious that the first group profited from the extremely efficient search for counterexamples, which required in average only about 3 membership queries. In fact, the algorithms in the first group executed 50 times as many approximative equivalence queries as the algorithms of the second group.

2. The impact of the continuous equivalence queries, which make only a difference of 1,8% in the ZULU ranking (E.H.Blocking vs Random), make about 6% in the number of detected states, which is a lot, when considering the typical behavior of Angluin-style learning. Just consider the only 3% difference in the number of detected states between the Random options of the first group and the second group. In the ZULU rating they make a difference of 19%.

3. Despite the extremely different distribution of the allowed 8101 membership queries, the average number of membership queries required to obtain a new state seem quite similar. They range between 23 and 26. One should keep in mind, however, that the difficulty to find a new state strongly increases in the course of the algorithm. Thus having detected 8% more states is much.

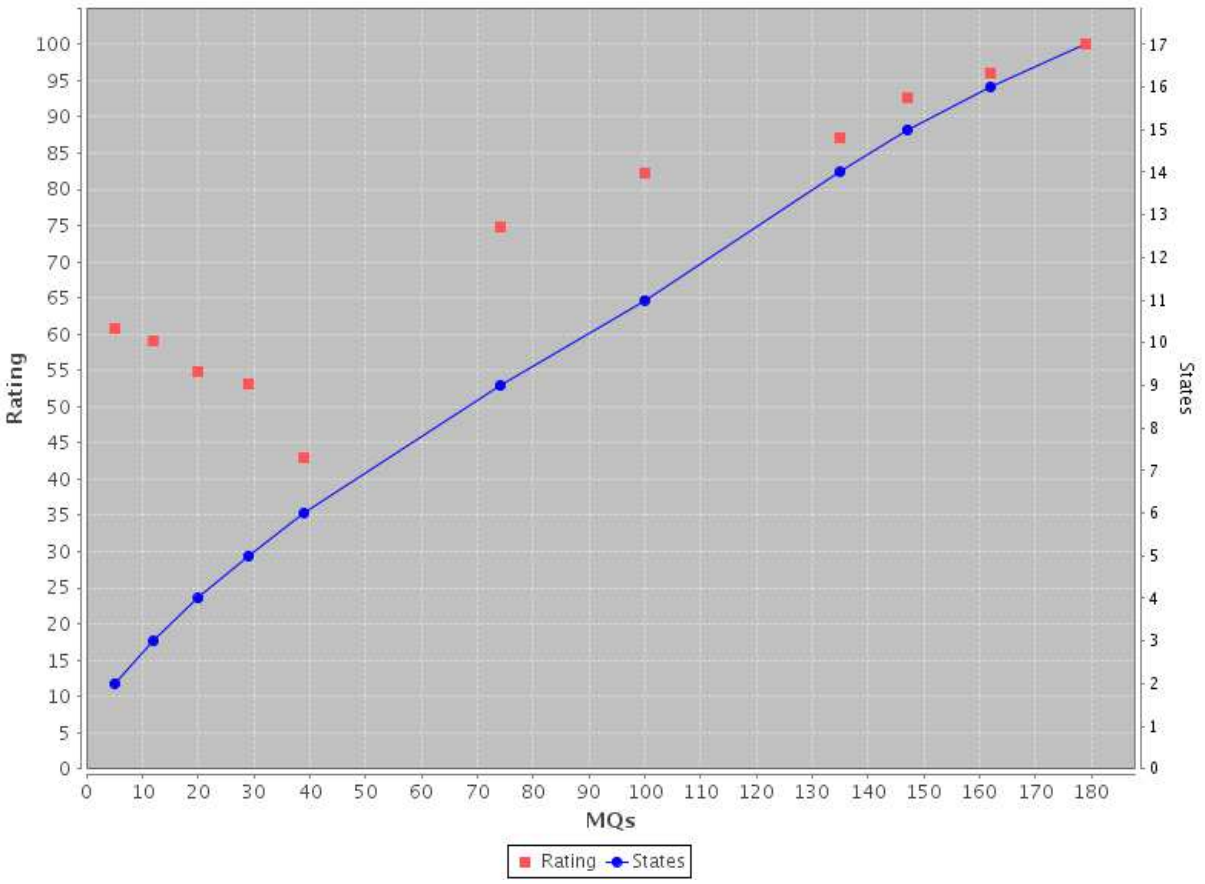

Figure 3.4: ZULU Rating for all hypotheses in a learning attempt, Problem 85173129

4. The ZULU ranking seems to increase with the number of states. This interpretation is, however, wrong, as can be seen in Fig. 3.4 (for ZULU problem 85173129), where in the beginning, the quality of prediction drops from 60 to almost only 40 per cent! For the already discussed example (49763507) this effect is not as strong. As we will discuss later, this effect, which shows here up already for randomly generated systems (in fact, we observed it in almost all Problems we treated as part of the ZULU challenge), gets worse when systems are man-made.

5. There is a part of the learning process, where small improvements make a big difference in the ZULU ranking (e.g. for the two random versions, 13 more states made a difference of 19% in the ZULU score). This is the part after the chaotic behavior at the beginning and before the final phase, where progress gets exponentially harder.

The ZULU organizers also observed that the top teams had quite similar scores. Looking at Table 3.2 this is not too surprising, as much progress is needed at the end to achieve even marginal improvements for the ZULU rating (20 more in this stage very hard to detect states for 1,8% in the ZULU rating: see E.H.Blocking vs. Random of the first group). It was therefore decided to further reduce the number of allowed membership queries. This certainly diversified the results. However, it also increased the risk of algorithms being punished by the quality drop described under item 4 above.

We consider the ZULU competition as milestone for advancing the state of the art for practical learning. Still, we see a high potential for improvement.

4 A Compact and Intuititive Rich Automaton Model

4.1 Introduction

Automata models that process words or trees over infinite alphabets are becoming increasingly important in many areas, including specification, verification, and testing (e.g., [14, 89]), databases, user modeling [27], etc. Often, such models have the form of a finite control structure, augmented by a finite set of registers (aka. state variables), which processes input symbols using a predefined set of operations (tests and updates) over input data and registers. Various such automata have long been used for specification and verification [14]. They have also received attention from a language-theoretic perspective, where decision problems and connections with logics have been studied (e.g., [32, 97]).

A crucial operation in the manipulating of automata is to minimize them and put them onto a canonical form. Minimization onto a canonical form is heavily used in verification, equivalence, and refinement checking, e.g., using (bi)simulation based criteria [69, 87], and it is the central principle underlying many techniques in automata learning (aka. regular inference) that construct minimal finite automata from a finite sample of accepted and rejected words [16, 48, 93]. For finite automata, there are standard algorithms for determinization and minimization, based on the Myhill-Nerode theorem [85, 57]. However, it has proven nontrivial to carry over such constructions to automata models over infinite alphabets. For instance, it has proven quite difficult to define minimal canonical forms for timed automata, Büchi automata, and other forms of automata. One complication has been that nondeterministic automata are typically more expressive than deterministic ones for most extensions of finite-state automata [86]. But even for the deterministic fragments, there are few definitions of minimal canonical forms.

In this chapter, we present a novel form of register automata, which also has an intuitive and succinct minimal canonical form, and which can be derived from a Nerode-like right congruence. By register automata, we mean automata with a finite control structure, equipped with a finite set of registers, that process words over an infinite alphabet consisting of terms with parameters from an infinite domain. When processing a next input symbol, the automaton can compare the parameters of the symbol with the contents of its registers to determine how to update registers and change control location. A similar class of automata has been considered by Kaminiski and Francez [68], who later provided a Myhill-Nerode theorem for it [46], later sharpened into a minimal form by Benedikt et al. [20] (note that these works use the term “register automata” with a more restrictive meaning). In contrast, our canonical form significantly improves on the above results by being significantly more compact and also significantly more intuitive. Both these properties are very important in many applications. For instance, in automata learning, the complexity of the learning procedure directly depends on the size of the minimal canonical form of the recognizing automaton [16, 93]; when constructing specifications in the form of automata, e.g., from requirement scenarios [55, 105], it is important that the resulting automata be intuitive, and also maintain an intuitive connection to the provided scenarios.

For our form of register automata, we present a Myhill Nerode-like theorem, from which minimal automata can be constructed, and an algorithm for minimization. Our form of register automata is as expressive as general deterministic Register Automata, but we show that our form can be exponentially more succinct than the variants and canonical forms defined by Kaminski and Francez [68, 46], and by Benedikt et al. [20]. Our definition is based on some novel ideas.

- In transitions, our automata only compare registers to input parameters if they are essential to whether the input word will be accepted or rejected. In contrast, the canonical register automata in [46, 20] must always consider each possible equality between an input parameter and a register as separate cases, which often causes a blow-up in the number of control states. In the automata of [46, 20], two registers may not contain the same data value, which sometimes forces redundant tests leading to a blow-up of control locations.

- Our automata are based on the principle of representing a language concisely, in terms of patterns of constraints on data values that cause a word to be rejected or accepted. Each such pattern (which is a particular combination of equalities between data values), is included in the representation only

if the verdict by the automaton (accept/reject) is different from slightly less constrained combinations. In other words, we present a language in terms of a number of particular cases, which are handled differently from more general case; these particular cases can be hierarchically ordered. This succinctness is very useful when specifying a language in terms of a number of representative words. For instance, when specifying a behavior by means of a number of scenarios, it is important to be able to provide a small set of scenarios, which capture the "essence" of the behavior. The language is then obtained by generalizing from these essential scenarios, letting the acceptance or rejection of a word be determined by that of its closest weaker essential word.

By a non-technical analogy, we could compare the difference between the automata of [46, 20] and ours to the difference between the region graph and zone graph constructions for timed automata. The region graph considers all possible combinations between constraints on clock values, be they relevant or not, whereas the zone graph construction aims to consider only relevant constraints. The analogy is not perfect, however, since (as we will prove) our automata are always more succinct than those of [46, 20].

Related Work.

Generalizations of regular languages to infinite alphabets have been studied for some decade. Kaminski and Francez [68] introduced finite memory automata (FMA) that recognize languages with infinite input alphabets. Data languages are introduced in [32] as languages over sequences of pairs (a, d) where a is a label from a finite domain and d is a data value from an infinite domain. A number of automata have been suggested (pebble automata, class memory automata, data automata) that accept different flavours of data languages differing in expressiveness (see [97, 28] for an overview). Most of the formalisms recognise data languages that are invariant under permutations on the data domain. In [31] a logical characterization of data languages is given alongside a transformation from logical description to automata. In [30] a solution for satisfiability of data languages that can be described using a first order logic fragment with variables is presented.

While most of the work focus on non-deterministic automata and are concerned with closedness properties and expressiveness results, we are interested in canonical automata (which are typically deterministic) and Myhill-Nerode theorem for data languages. Such a theorem would open data languages to applications in, e.g., Angluin-style active learning. This form of learning is now receiving increasing attention in the testing and verification community, e.g., for regression testing of telecommunication systems [54, 62], for integration testing [53, 73], security protocol testing [100], and for combining conformance testing and model checking [88, 52].

Only in [46, 20] a Myhill-Nerode theorem for a special kind of data automata is presented. However, the used automaton model is quite restrictive, state variables are stored in a queue-like data structure: new data values can be added only to the end of the queue, and data values which must no longer be remembered are dropped from the queue. In the automaton model we propose, we do not order the state variables in this way, we can represent some data languages more succinctly; indeed, in some cases, our model is even exponentially more succinct.

Targeting applications like learning, we want to model reactive systems with complex parameterized action (e.g., web services) by data languages. Thus, we will extend the notion of data languages to sequences of pairs $(a, \overline{d})$, where $\overline{d}$ is a vector of data values from an infinite domain.

Organization.

In the next section we give an initial motivating example, to provide some intuition behind our contribution, before we define the notion of data language and the general form of register automata we will use in Section 4.3. In Section 4.4 we present a novel, decision tree-like representation of data languages that allows us to capture exactly the relevant relations between data values. This representation will be used in Section 4.5 to define a special form of register automata (called DCDTAs) that builds on this representation. In Section 4.6 we formulate a Myhill-Nerode-like theorem for data languages, which provides a concise canonical form for our automata. The Nerode-theorem also provides the basis for the

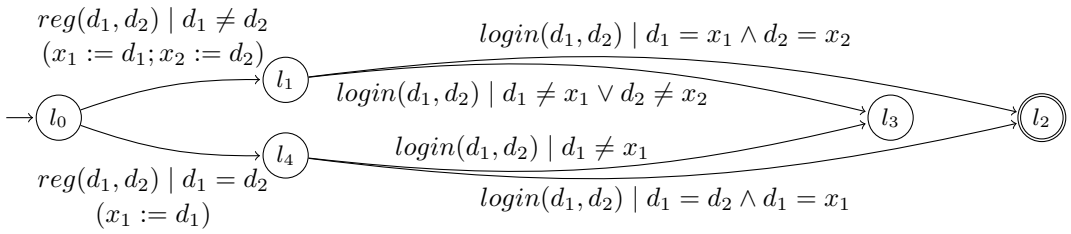

Figure 4.1: A canonical automaton, recognizing $\mathcal{L}_{login}$

minimization algorithm presented in Section 4.7. Finally, in Section 4.8 we relate the canonical form we suggest in this chapter to previously suggested canonical forms before we conclude in Section 4.9.

4.2 An Illustrating Example

In this section, we provide an overview of our automaton model and our main ideas using a simple running example. Our example is a language, which represents successful user authentications to a fictitious system. The language consists of sequences of symbols of form reg(d_1, d_2) and login(d_1, d_2), where d_1 and d_2 are data values from some infinite set (e.g., integers or strings). Intuitively, the first parameter d_1 represents a user name, while the second parameter d_2 represents a password. The language $\mathcal{L}_{login}$ of successful user authentications is the set of words

$$\mathcal{L}_{login} = \{\text{reg}(d_1, d_2)\text{login}(d_3, d_4) \mid d_1 = d_3 \wedge d_2 = d_4\}.$$

Let us consider the problem of defining a class of canonical register automata, in which languages such as $\mathcal{L}_{login}$ can be represented succinctly. It seems reasonable that any automaton, after reading the first symbol reg(d_1, d_2) must remember the two data values d_1 and d_2 in state variables. Previous proposals (by Kaminski and Francez [68, 46], and by Benedikt et al. [20]) obtain canonicity by letting each possible valuation of tests available to the automaton (in this case only for equality) correspond to a separate state. Consequently, symbols of form reg(d_1, d_2) take the automaton to (at least) two different states, depending on whether $d_1 = d_2$ or not. In this case, we need at most one state variable for each distinct value in the set $\{d_1, d_2\}$ The automaton can then look as in Figure 4.1.

In contrast, our automata only perform tests on input parameters if they are relevant to whether the input word will be accepted or rejected. In the above example, the test $d_1 = d_2$ on the parameters of reg(d_1, d_2) is irrelevant for $\mathcal{L}_{login}$. For this reason, our version of canonic automata will not perform this test. Our automaton will store both d_1 and d_2, in two different variables, regardless of whether they are equal or not, resulting in a more natural and more succinct automaton, as in Figure 4.2.

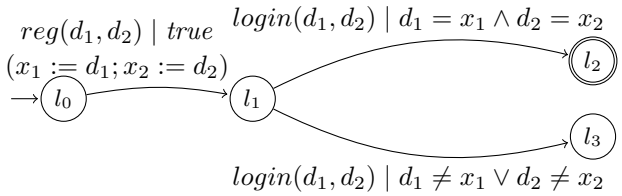

Figure 4.2: Our canonical automaton, recognizing $\mathcal{L}_{login}$

In this particular example, our automaton has only one state less than the Figure 4.1 automaton , but in general the cost of keeping track of all possible relations between state variables can be exponential, as shown in Proposition 11.

In order to distinguish "relevant" from "irrelevant" comparisons, we will slightly change our representation of languages. Instead of considering words of "instantiated" symbols (such as reg$(\texttt{Bob}, 14)$) we

represent words as patterns of constraints on data values that cause a word to be rejected or accepted. Each such pattern (which is a particular combination of equalities between data values), is included in the representation only if the verdict by the automaton (accept/reject) is different from slightly less constrained combinations. The language $\mathcal{L}_{login}$ can then be succinctly represented by stating that the pattern $\{\mathsf{reg}(d_1, d_2)\mathsf{login}(d_3, d_4) \mid true\}$ is rejected, whereas the pattern $\{\mathsf{reg}(d_1, d_2)\mathsf{login}(d_3, d_4) \mid d_1 = d_3 \wedge d_2 = d_4\}$ is a weakest exception to the previous pattern, and accepted.

In our work, we show that any language can be represented by minimal set of such patterns in a canonical way, using a decision tree-like structure. Our automata can then be understood as "foldings" of such decision trees, and are minimal and canonical for the same reasons as the trees.

4.3 Data Languages and Register Automata

We begin by assuming a domain D and a set of *action types* I. Each action type α has a certain *arity*, which determines how many parameters it takes from the domain D.

A *data symbol* is a term of the form $\alpha(d_1, \ldots, d_n)$, where α is an action type with arity n, and $d_1, \ldots, d_n$ are data values in D. We define Σ_I^D as the set of data symbols of the form $\alpha(d_1, \ldots, d_n)$, where $\alpha \in I$.

We also assume a set of binary predicate symbols. In this chapter, we will only use binary equality $=$.[1]

A *data word* is a sequence of data symbols. We define an equivalence $\simeq$ on the set $(\Sigma_I^D)^*$ of data words. We are interested only in the relevant patterns between data values. Thus, for two data words u and v, $u \simeq v$ iff there is some permutation of D such that u is equal to the component-wise application of this permutation to the data values in v. Turning to our running example $\mathcal{L}_{login}$, this means that

$$\mathsf{reg}(\mathtt{Ann}, 78)\mathsf{login}(\mathtt{Ann}, 78) \simeq \mathsf{reg}(\mathtt{Bob}, 18)\mathsf{login}(\mathtt{Bob}, 18)$$

but

$$\mathsf{reg}(\mathtt{Ann}, 78)\mathsf{login}(\mathtt{Ann}, 78) \not\simeq \mathsf{reg}(\mathtt{Bob}, 18)\mathsf{login}(\mathtt{Bob}, 99).$$

A *data language* $\mathcal{L}$, finally, is a set of data words closed under $\simeq$, i.e., such that $w \in \mathcal{L}$ and $w \simeq w'$ implies $w' \in \mathcal{L}$.

We see that our definition of data languages assumes that all data values are treated symmetrically. This is a natural assumption, e.g., in (web) services that expose the same behavior to every user, or network protocols that require passing along a session identifier.

We can now present an automaton model that recognizes data languages. Assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, and a finite set of *variables* (or registers), ranged over by $x_1, x_2, \ldots$. A *parameterized symbol* is a term of form $\alpha(p_1, \ldots, p_n)$, consisting of an action type α and formal parameters $p_1, \ldots, p_n$ (respecting the arity of α). A *guard* is a boolean formula in DNF of atomic predicates of form $z_i = z_j$ or $z_i \neq z_j$, where each of z_i and z_j is either a formal parameter or a variable. We write $\overline{p}$ for $p_1, \ldots, p_n$, $\overline{d}$ for $d_1, \ldots, d_n$, and $\overline{x}$ for $x_1, \ldots, x_k$.

Definition 2. A *Register Automaton* (RA) is a tuple $\mathcal{A} = (I, L, l_0, T, \lambda)$, where

- I is a finite set of action types.
- L is a finite set of *locations*; with each location l, we associate a tuple $X(l)$ of variables.
- $l_0 \in L$ is the *initial location*, with $X(l_0)$ being the empty tuple,
- T is a finite set of *transitions*, each of which is of form $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle$, where l is a *source location*, $\alpha(\overline{p})$ is a parameterized symbol, g is a guard over $\overline{p}$ and $X(l)$, π (the *assignment*) is a mapping from $X(l')$ to $X(l) \cup \overline{p}$, and l' is a *target location*,
- $\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject). □

[1] In future work, we plan to cover simple extensions, such as a total order $<$, or membership $\in$.

We write $l \xrightarrow{\alpha(\overline{p});g/\pi} l'$ to denote that $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle \in T$. The RA should be *completely specified*, meaning for any location l and input action α, the disjunction of all guards g in transitions of form $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle$ in T is equivalent to $true$. The RA is *deterministic* (called a DRA) if for any location l and input action α, the conjunction $g_1 \wedge g_2$ is unsatisfiable whenever $\langle l, \alpha(\overline{p}), g_1, \pi_1, l'_1 \rangle$ and $\langle l, \alpha(\overline{p}), g_2, \pi_2, l'_2 \rangle$ are different transitions from the same location with the same input action in T.

Let us define the semantics of an RA $\mathcal{A} = (I, L, l_0, T, \lambda)$. For a tuple X of variables, a X-valuation is an assigment $\sigma^X : X \mapsto \mathcal{D}$ of data values to the variables in X. Valuations are extended to predicates and guards in the natural way. A *state* of $\mathcal{A}$ is a pair $\langle l, \sigma^{X(l)} \rangle$ where $l \in L$ and $\sigma^{X(l)}$ is a $X(l)$-valuation. The *initial state* is $\langle l_0, \sigma^{\emptyset} \rangle$. A *step* of $\mathcal{A}$ is a triple, written

$$\langle l, \sigma^{X(l)} \rangle \xrightarrow{\alpha(\overline{d})} \langle l', \sigma^{X(l')} \rangle$$

such that T contains a transition $\langle l, \alpha(\overline{d}), g, \pi, l' \rangle$ such that $\sigma^{X(l)}(g[\overline{d}/\overline{p}])$ is true, and such that $\sigma^{X(l')}(x_i) = \sigma^{X(l)}(\pi(x_i))$ if $\pi(x_i) \in X(l)$ and $\sigma^{X(l')}(x_i) = d_j$ if $\pi(x_i) = p_j$ for p_j in $\overline{p}$. A *run* of $\mathcal{A}$ over a data word $w = \alpha_1(\overline{d}_1) \cdots \alpha_k(\overline{d}_k)$ is a sequence of steps

$$\langle l_0, \sigma^{\emptyset} \rangle \xrightarrow{\alpha_1(\overline{d}_1)} \langle l_1, \sigma^{X(l_1)} \rangle \xrightarrow{\alpha_2(\overline{d}_2)} \cdots \xrightarrow{\alpha_k(\overline{d}_k)} \langle l_k, \sigma^{X(l_k)} \rangle \ ,$$

which we sometimes denote $\langle l_0, \sigma^{\emptyset} \rangle \xrightarrow{w} \langle l_k, \sigma^{X(l_k)} \rangle$. Such a run is *accepting* if $\lambda(l_k) = +$, otherwise it is *rejecting*. A data word is *accepted* by $\mathcal{A}$ if it has an accepting run over it. The language recognized by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$ is the set of data words that it accepts.

A state $\langle l, \sigma \rangle$ is *reachable* if $\langle l_0, \sigma^{\emptyset} \rangle \xrightarrow{w} \langle l, \sigma \rangle$ for some data word w.

An example of a DRA accepting the language $\mathcal{L}_{login}$ is shown in Figure 4.2.

4.4 Constrained Word Representation of Languages

As stated before, we will define a concise automaton model, which can be seen as a restricted form of DRAs. In our model, the possible steps between states are not represented by a set of transitions, as in DRAs, but by a decision tree-like structure for each location. We call our model DCDTA (Determinate Constraint Decision Tree Automata). DCDTAs can easily be transformed to DRAs by "flattening" the decision trees into transitions, but we will for this section concentrate on explaining decision trees, since they convey our ideas and definitions in a more natural manner.

Recall that a parameterized symbol is of form $\alpha(p_1, \ldots, p_n)$ where $p_1, \ldots, p_n$ are formal parameters. Let Σ_I^P denote the set of parameterized symbols $\alpha(\overline{p})$ with $\alpha \in I$. A *parameterized word* (or just *word*) is a sequence of parameterized symbols in which all formal parameters are distinct; we use $\overline{p_w}$ to denote the ordered sequence of formal parameters in w. A *constraint* is a conjunction of atomic predicates over formal parameters (i.e., of form $p_i = p_j$); note that a constraint does not contain negated predicates. A *constrained word* is a pair $\langle w, \phi \rangle$ consisting of a parameterized word w and a constraint ϕ over the formal parameters $\overline{p_w}$ of w. For a set $W \subseteq (\Sigma_I^P)^*$ of parameterized words, let $\Phi[W]$ denote the set of constrained words $\langle w, \phi \rangle$ with $w \in W$.

Let $\overline{p_w}$ be the ordered sequence $p_1, \ldots, p_n$. We require that constraints be written on a unique normal form, namely as an ordered conjunction $p_{i_1} = p_{j_1} \wedge p_{i_2} = p_{j_2} \wedge \cdots \wedge p_{i_k} = p_{j_k}$, where

1. each atomic predicate $p_{i_l} = p_{j_l}$ is an ordered pair with $i_l < j_l$, and
2. whenever $p_i = p_j$ appears before $p_{i'} = p_{j'}$, then $i \neq i'$ and $j < j'$.

Each constraint ϕ then has a unique normal form. By the associativity of conjunction, we can write it as an ordered list $p_{i_1} = p_{j_1} \wedge p_{i_2} = p_{j_2}, \cdots \wedge p_{i_k} = p_{j_k}$ with $j_1 < \cdots < j_k$. In the following, whenever we write a constraint as $\theta \wedge \psi$, we assume θ and ψ are normalized constraints such that their concatenation $\theta \wedge \psi$ is immediately on normal form. For the general case, we let $\phi \sqcap \phi'$ denote the normalized constraint equivalent to the conjunction of ϕ and ϕ'. We use $true$ to denote the empty constraint.

Let $\phi \sqsubseteq \phi'$ denote that ϕ' implies ϕ. Define a strict partial order $<$ on (normalized) constraints, by letting $\phi < \phi'$ denote that ϕ is of form $\theta \wedge \psi$, and ϕ' is of form $\theta \wedge p_i = p_j \wedge \psi'$ for some longest common prefix θ, such that $j < j'$ for any atomic predicate $p_{i'} = p_{j'}$ in ψ. Let $\phi \leq \phi'$ denote that $\phi < \phi'$ or $\phi = \phi'$. We observe that $\phi \sqsubseteq \phi'$ implies $\phi \leq \phi'$, and that $true$ is the smallest constraint wrt. to both $\sqsubseteq$ and $<$. For any parameterized word w, let $\langle w, \phi\rangle \sqsubseteq \langle w, \phi'\rangle$ denote that $\phi \sqsubseteq \phi'$, and let $\langle w, \phi\rangle < \langle w, \phi'\rangle$ denote that $\phi < \phi'$.

A data word $\alpha_1(\overline{d}_1)\cdots\alpha_k(\overline{d}_k)$ *satisfies* a constrained word $\langle \alpha_1(\overline{p}_1)\cdots\alpha_k(\overline{p}_k), \phi\rangle$ if the data word and the parameterized word have the same sequence of action types, and d_{i_p} is equal to d_{j_q} whenever $p_{i_p} = p_{j_q}$ is a conjunct in ϕ. For a data word $\alpha_1(\overline{d}_1)\cdots\alpha_k(\overline{d}_k)$, define *cw*$[\alpha_1(\overline{d}_1)\cdots\alpha_k(\overline{d}_k)]$ as the strongest constrained word that it satisfies. We can then represent a data language $\mathcal{L}$ by the unique mapping $\lambda : \Phi[(\Sigma_I^P)^*] \mapsto \{+,-\}$ from the set of constrained words to $\{+,-\}$, such that $\lambda($*cw*$[\alpha_1(\overline{d}_1)\cdots\alpha_k(\overline{d}_k)]) = +$ iff $\alpha_1(\overline{d}_1)\cdots\alpha_k(\overline{d}_k) \in \mathcal{L}$. It is clear that this correspondence is one-to-one, and therefore we will from now on consider mappings from constrained words to $\{+,-\}$ instead of data languages.

For any set W of parameterized words and finite set R, we will now define a succinct representation for mappings $\lambda : \Phi[W] \mapsto R$ by a decision tree-like structure, which only defines λ on constrained words that precisely capture exceptions from the value of λ on "slightly weaker" constrained words. The value of λ is then implicitly defined on any constrained word, by generalizing its value from "strongest" slightly weaker constrained word which has been explicitly included in the domain of λ. It should be noted that there may be several "slightly weaker" constrained words, and therefore, for determinacy, we require that λ has the same value whenever a constrained word may have several "slightly weaker" words. Moreover, our decision trees can be defined in the same way for any finite range, whence we here give the definition for an arbitrary finite set R, which will typically be $\{+,-\}$.

We say that $\langle u, \theta\rangle$ is a *prefix* of $\langle w, \phi\rangle$, denoted $\langle u, \theta\rangle \propto \langle w, \phi\rangle$, if u is a (not necessarily proper) prefix of w and θ is a (not necessarily proper) prefix of ϕ (recall that constraints are ordered lists). Let W be a set of parameterized words. A set $\Phi \subseteq \Phi[W]$ of constrained words is *prefix-closed* wrt. W if $\langle w, \phi\rangle \in \Phi$, $\langle u, \theta\rangle \in \Phi[W]$, and $\langle u, \theta\rangle \propto \langle w, \phi\rangle$ implies $\langle u, \theta\rangle \in \Phi$. It is *suffix-closed* wrt. W if $\langle u, \phi\rangle \in \Phi$, $w \in W$, and $u \propto w$ implies $\langle w, \phi\rangle \in \Phi$.

Definition 3 (Constraint Decision Tree)**.** *Let W be a set of parameterized words, and let R be an arbitrary finite set. A* constraint decision tree *(CDT) $\mathcal{T}$ from W to R is a pair $\langle Dom(\mathcal{T}), \lambda_\mathcal{T}\rangle$ where $Dom(\mathcal{T}) \subseteq \Phi[W]$ is a prefix-closed and suffix-closed (wrt. W) set of constrained words which contains $\langle w, true\rangle$ for each $w \in W$, and $\lambda_\mathcal{T} : Dom(\mathcal{T}) \mapsto R$ is a mapping from $Dom(\mathcal{T})$ to R.* □

A CDT $\mathcal{T}$ defines a mapping from all of $\Phi[W]$ to R, which is obtained by extending the domain of $\lambda_\mathcal{T}$ as follows. For a CDT, we define a relation $\preceq_\mathcal{T} \subseteq Dom(\mathcal{T}) \times \Phi[W]$ between constrained words in $\mathcal{T}$ and arbitrary constrained words. Intuitively, $\langle w, \phi'\rangle \preceq_\mathcal{T} \langle w, \phi\rangle$ means that $\langle w, \phi'\rangle$ is a "slightly weaker" constrained word, and we will use it to define the value of $\lambda_\mathcal{T}$ for $\langle w, \phi\rangle$. . We say that $\langle w, \phi'\rangle \preceq_\mathcal{T} \langle w, \phi\rangle$ iff $\langle w, \phi'\rangle$ is a maximal (wrt. $<$) constrained word in $Dom(\mathcal{T})$ with $\langle w, \phi'\rangle \sqsubseteq \langle w, \phi\rangle$. (In other words, $\langle w, \phi'\rangle < \langle w, \phi''\rangle$ implies that either $\langle w, \phi''\rangle \notin Dom(\mathcal{T})$ or $\langle w, \phi''\rangle \not\sqsubseteq \langle w, \phi\rangle$.)

Because there can be several such "slightly weaker" constrained words, we define a *determinate* CDT (or DCDT) as a CDT where whenever $\langle w, \phi'\rangle \preceq_\mathcal{T} \langle w, \phi\rangle$ and $\langle w, \phi''\rangle \preceq_\mathcal{T} \langle w, \phi\rangle$, $\lambda_\mathcal{T}(\langle w, \phi'\rangle) = \lambda_\mathcal{T}(\langle w, \phi''\rangle)$. Any DCDT $\mathcal{T}$ from W to R defines the function $\lceil\lambda_\mathcal{T}\rceil^\mathcal{T} : \Phi[W] \mapsto R$ so that $\lceil\lambda_\mathcal{T}\rceil^\mathcal{T}(\langle w, \phi\rangle) = \lambda_\mathcal{T}(\langle w, \phi'\rangle)$ whenever $\langle w, \phi'\rangle \preceq_\mathcal{T} \langle w, \phi\rangle$.

Intuitively, if $\langle w, \phi'\rangle$ is "slightly weaker" than $\langle w, \phi\rangle$ (i.e., $\langle w, \phi'\rangle \preceq_\mathcal{T} \langle w, \phi\rangle$), then $\langle w, \phi'\rangle$ can be viewed as the "run" or "path" in $Dom(\mathcal{T})$ which is used to decide the value of $\lceil\lambda\rceil^\mathcal{T}(\langle w, \phi\rangle)$. The run or path $\langle w, \phi'\rangle$ can be inductively constructed from $\langle w, \phi\rangle$ as the limit of a sequence of prefixes in $Dom(\mathcal{T})$, as follows.

- The initial prefix is $\langle w, true\rangle$.
- If $\langle w, \theta\rangle$ is a prefix, then the next prefix is obtained by adding the atomic predicate $p_i = p_j$ with the smallest j, which is implied by ϕ, and such that $\langle w, \theta \wedge p_i = p_j\rangle$ is in $Dom(\mathcal{T})$ (recall that we assume $\theta \wedge p_i = p_j$ to be in normal form).
- If there is no such next prefix following $\langle w, \theta\rangle$, then it follows by our definitions that $\langle w, \phi'\rangle \preceq_\mathcal{T} \langle w, \phi\rangle$.

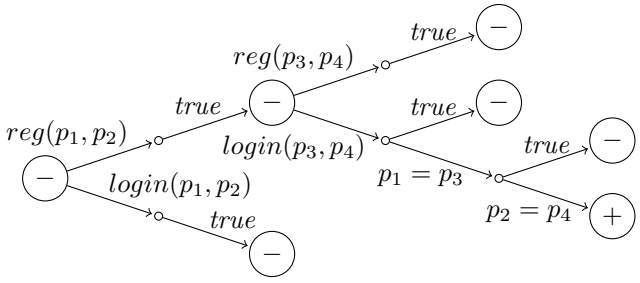

Figure 4.3: Prefix of minimal DCDT for $\mathcal{L}_{login}$

Intuitively, $\langle w, \phi' \rangle$ is a "path" in the $Dom(\mathcal{T})$, which after each prefix of ϕ' branches on the next relevant atomic constraint in ϕ.

We provide an example of a decision tree for the language $\mathcal{L}_{login}$ in Figure 4.3. We can see that an example run which is accepted (reaching $+$) necessarily starts with reg(p_1, p_2). We then need to process login(p_3, p_4) with the constraint that $p_1 = p_3$ and $p_2 = p_4$ in order to reach an accepting location. Replacing one of these constraints with $true$ will invariably result in nonacceptance.

An important property of DCDTs, which is crucial for obtaining canonicity of our automaton model, is that each mapping $\lambda : \Phi[W] \mapsto R$ can be represented by a unique minimal DCDT.

Theorem 4 (Minimal DCDT)**.** *For any function $\lambda : \Phi[W] \mapsto R$ there is a unique minimal DCDT $\mathcal{T}$ from W to R such that $\lceil \lambda_{\mathcal{T}} \rceil^{\mathcal{T}} \equiv \lambda$.* □

This unique minimal DCDT is called the *minimal DCDT for* λ. Minimality of $\mathcal{T}$ means that whenever $\lceil \lambda_{\mathcal{T}'} \rceil^{\mathcal{T}'} \equiv \lambda$ for some DCDT $\mathcal{T}'$, then $Dom(\mathcal{T}) \subseteq Dom(\mathcal{T}')$.

Proof Assume a function $\lambda : \Phi[W] \mapsto R$. We need to show that there is a minimal subset U of $\Phi[W]$, such that we can take $Dom(\mathcal{T})$ to be U and $\lambda_{\mathcal{T}}$ to be the restriction of λ to U. Our proof shows how to construct such a minimal set incrementally, using a set of saturation rules. Each rule requires a certain constrained word to be in U if some other constrained word is in U. The rules are as follows.

- U must contain the constrained word $\langle w, true \rangle$ for each $w \in (\Sigma_I^P)^*$.
- U must be prefix- and suffix-closed wrt. W.
- Assume that U contains a constrained word $\langle w, \phi' \rangle$ and $\langle w, \phi \rangle \in \Phi[W]$ is a constrained word such that (1) $\langle w, \phi' \rangle \sqsubseteq \langle w, \phi \rangle$, and (2) $\lambda(\langle w, \phi' \rangle) \neq \lambda(\langle w, \phi \rangle)$, and (3) whenever $\langle w, \phi' \rangle < \langle w, \phi'' \rangle < \langle w, \phi \rangle$ and $\langle w, \phi'' \rangle \sqsubseteq \langle w, \phi \rangle$ for some $\langle w, \phi'' \rangle$, then $\lambda(\langle w, \phi' \rangle) = \lambda(\langle w, \phi'' \rangle)$. Then U must contain the shortest prefix of $\langle w, \phi \rangle$ which is not a prefix of $\langle w, \phi' \rangle$, i.e., if ϕ' is of form $\theta \wedge \psi'$ and ϕ is of form $\theta \wedge p_i = p_j \wedge \psi$ where $p_i = p_j$ is not in ψ', then U must contain $\langle w, \theta \wedge p_i = p_j \rangle$.

The motivation for the two first rules is obvious. For the third rule, we observe that for any $\mathcal{T}'$ where $Dom(\mathcal{T}')$ does not satisfy the third rule we have $\lceil \lambda_{\mathcal{T}'} \rceil^{\mathcal{T}'} (\langle w, \phi \rangle) \neq \lambda(\langle w, \phi \rangle)$, where $\langle w, \phi \rangle$ is as in the third rule. To conclude the proof, it is easy to see that (1) any DCDT $\mathcal{T}'$ with $\lceil \lambda_{\mathcal{T}'} \rceil^{\mathcal{T}'} \equiv \lambda$ must satisfy the saturation rules, and that (2) if U satisfies all saturation rules, and we choose $Dom(\mathcal{T})$ to be U and $\lambda_{\mathcal{T}}$ to be the restriction of λ to U, then $\lceil \lambda_{\mathcal{T}} \rceil^{\mathcal{T}} \equiv \lambda$. It then follows by standard arguments that there is a unique minimal set U which satisfies all the saturation rules.

4.5 A Compact Automaton Model

From the minimal DCDT described in Theorem 4, we can now proceed to define a special form of DRAs, which can be thought of as folding a DCDT into an automaton. Before we can do that, we must generalize

the notion of constrained words and decision trees to handling location variables. We must also define how to split constrained words into prefixes and suffixes.

Consider a constrained word $\langle w, \phi \rangle$, where w is a concatenation uv. We can make a corresponding split of ϕ as $\theta \wedge \psi$, where the right-hand sides of atomic predicates in θ are in $\overline{p_u}$, and the right-hand sides of atomic predicates in ψ are in $\overline{p_v}$. Then $\langle u, \theta \rangle$ (the prefix) is a normal constrained word, but $\langle v, \psi \rangle$ (the suffix) is not, since ψ refers to parameters that are not in v.

We proceed to define the *potential* of a constrained word $\langle u, \theta \rangle$, denoted $\Delta_{\langle u,\theta \rangle}$, as the set of formal parameters in u that do not occur as the left argument of any atomic predicate in θ. These can be constrained in atomic predicates when $\langle u, \theta \rangle$ is extended by a suffix. We distinguish between *concrete*, *abstract*, and *restricted* suffixes:

A *concrete* $\langle u, \theta \rangle$*-suffix* (or just concrete suffix) is a tuple $\langle v, \psi \rangle$, where ψ is a normalized constraint of atomic predicates of form $p_i = p_j$, where $p_i \in \Delta_{\langle u,\theta \rangle} \cup \overline{p_v}$ and $p_j \in \overline{p_v}$. For a prefix $\langle u, \theta \rangle$ and a concrete $\langle u, \theta \rangle$-suffix $\langle v, \psi \rangle$, we use $\langle u, \theta \rangle; \langle v, \psi \rangle$ to denote $\langle uv, \theta \wedge \psi \rangle$.

Assuming a set of variables, ranged over by $x_1, x_2, \ldots x_k$, an *abstract suffix* is a pair $\langle v, \psi(\overline{x}) \rangle$, where v is a parameterized word, and $\psi(\overline{x})$ is a conjunction of atomic predicates $z = p_j$, where $z \in \overline{x} \cup \overline{p_v}$ and $p_j \in \overline{p_v}$.

A *restricted suffix* is a pair $\langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$ where $\langle v, \psi(\overline{x}) \rangle$ is a suffix and ζ is a conjunction of equalities between the variables $\overline{x}$.

Just as for constrained words, the constraint $\psi(\overline{x})$ in (both concrete and abstract) suffixes should be normalized, as an ordered conjunction where the right-hand sides are parameters in strictly ascending order, and no variable or parameter occurs more than once as the left-hand side of an atomic constraint. The *potential* of a suffix $\langle v, \psi(\overline{x}) \rangle$, denoted $\Delta_{\langle v,\psi(\overline{x}) \rangle}$, is the set of variables in $\overline{x}$ and formal parameters in v that do not occur as the left argument of any atomic predicate in $\psi(\overline{x})$. If π is an injective mapping from $\overline{x}$ to $\Delta_{\langle u,\theta \rangle}$, we write $\pi(\langle v, \psi(\overline{x}) \rangle)$ for the concrete suffix obtained by replacing variables according to π, so that we can write $\langle u, \theta \rangle; \pi(\langle v, \psi(\overline{x}) \rangle)$ for $\langle uv, \theta \wedge \pi(\psi(\overline{x})) \rangle$.

Intuitively, abstract suffixes will be used in decision trees, and restricted suffixes will be used to denote a partially processed constrained word, where the conjunction ζ represents the equalities between variables that have been accumulated when processing some previous prefix. Note that any suffix $\langle v, \psi(\overline{x}) \rangle$ can be regarded as the restricted suffix $\langle true, \langle v, \psi(\overline{x}) \rangle \rangle$.

For a set $W \subseteq (\Sigma_I^P)^*$ of parameterized words, and a set of variables $\overline{x}$, let $\Phi(\overline{x})[W]$ denote the set of abstract suffixes $\langle v, \psi(\overline{x}) \rangle$ and $\Xi(\overline{x})[W]$ denote the set of restricted suffixes $\langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$, with $v \in W$ and variables in $\overline{x}$.

For an abstract suffix $\langle v, \psi'(\overline{x}) \rangle$ and a restricted suffix $\langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$, define $\langle v, \psi'(\overline{x}) \rangle \sqsubseteq \langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$ iff the conjunction of ζ and $\psi(\overline{x})$ implies $\psi'(\overline{x})$. Define the relation $<$, and the notion of prefix on abstract suffixes exactly as for constrained words.

We can then generalize the notion of CDTs to abstract suffixes in the following way:

Definition 5 (Suffix Constraint Decision Tree)**.** *Let W be a set of parameterized words, $\overline{x}$ be a set of variables, and let R be an arbitrary finite set. A* suffix constraint decision tree *(SCDT) $\mathcal{T}$ over $\overline{x}$ from W to R is a pair $\langle Dom(\mathcal{T}), \lambda_{\mathcal{T}} \rangle$ where $Dom(\mathcal{T})$ is a prefix-closed and suffix-closed (wrt. W) set of abstract $\overline{x}$-suffixes, which contains $\langle w, true \rangle$ for each $w \in W$, and $\lambda_{\mathcal{T}} : Dom(\mathcal{T}) \mapsto R$ is a mapping from $Dom(\mathcal{T})$ to R.* □

Continuing the analogy, an SCDT $\mathcal{T}$ over $\overline{x}$ defines a mapping from all of $\Phi(\overline{x})[W]$ to R, obtained by extending the domain of $\lambda_{\mathcal{T}}$ as for CDTs. Define the relation $\preceq_{\mathcal{T}}$ between suffixes and restricted suffixes by $\langle v, \psi'(\overline{x}) \rangle \preceq_{\mathcal{T}} \langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$ iff $\langle v, \psi'(\overline{x}) \rangle$ is a maximal (wrt. $<$) suffix in $Dom(\mathcal{T})$ with $\langle v, \psi'(\overline{x}) \rangle \sqsubseteq \langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$. For a restriction ζ, say that a SCDT $\mathcal{T}$ is *determinate* wrt. ζ (called ζ-DSCDT) if $\langle v, \psi'(\overline{x}) \rangle \sqsubseteq \langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$ and $\langle v, \psi''(\overline{x}) \rangle \sqsubseteq \langle \zeta, \langle v, \psi(\overline{x}) \rangle \rangle$ imply $\lambda_{\mathcal{T}}(\langle v, \psi'(\overline{x}) \rangle) = \lambda_{\mathcal{T}}(\langle v, \psi''(\overline{x}) \rangle)$. We have the analogue of Theorem 4.

Theorem 6 (Minimal DSCDT)**.** *For any function $\lambda : \Xi(\overline{x})[W] \mapsto R$ there is a unique minimal DSCDT $\mathcal{T}$ over $\overline{x}$ from W to R such that $\lceil \lambda_{\mathcal{T}} \rceil^{\mathcal{T}} \equiv \lambda$.* □

We are now able to define our automaton model.

Definition 7. A *CDT automaton* (CDTA) is a tuple $\mathcal{A} = (I, L, l_0, \mathcal{T}, \lambda)$, where I, L, l_0, and λ stand for the same as in the definition of DRAs, but where $\mathcal{T}(l)$ maps each location $l \in L$ to an SCDT $\mathcal{T}(l)$ over $X(l)$ from suffixes in $\Phi(X(l))[\Sigma_I^P]$ of form $\langle \alpha(\overline{p}), \psi(\overline{x})\rangle$ to pairs of form $\langle \pi, l'\rangle$ where $l' \in L$ and π is an injective mapping from $X(l')$ to $\Delta_{\langle \alpha(\overline{p}), \psi(\overline{x})\rangle}$. □

In order to define the semantics and determinacy of a CDTA, we need some intermediate definitions.

For a normalized constraint ψ over variables and parameters, we let κ_ψ denote the mapping which maps each variable in $\overline{x}$ and parameter in $\overline{p_v}$ to the unique variable or parameter, which is not the LHS of any equality, which it must be equal to. We can inductively define it by

- κ_{true} is the identity mapping,
- $\kappa_{(z=p_i \wedge \psi)} = \kappa_\psi[z \mapsto \kappa_\psi(p_i)]$

Thus, for a suffix $\langle v, \psi(\overline{x})\rangle$, $\kappa_{\psi(\overline{x})}$ is a mapping from $\overline{x} \cup \overline{p_v}$ to $\Delta_{\langle v, \psi(\overline{x})\rangle}$. We extend $\kappa_{\psi(\overline{x})}$ to map boolean combinations of equalities between variables and parameters to boolean combinations of equalities in the natural way.

Given an SCDT $\mathcal{T}$, define for each $\langle v, \psi(\overline{x})\rangle \in Dom(\mathcal{T})$ the set of *implicit inequalities* of $\langle v, \psi(\overline{x})\rangle$, denoted $\textit{ineqs}_{\mathcal{T}}(\langle v, \psi(\overline{x})\rangle)$ as the negation of all atomic predicates $z = p_j$ for which there is some $\langle u, \theta \wedge z = p_j\rangle \in Dom(\mathcal{T})$ such that $\langle u, \theta\rangle$ is a prefix of $\langle v, \psi(\overline{x})\rangle$ and $\langle v, \psi(\overline{x})\rangle < \langle u, \theta \wedge z = p_j\rangle$. Intuitively, $z = p_j$ is an equality not satisfied when processing $\langle v, \psi(\overline{x})\rangle$ by $\mathcal{T}$.

We can now annotate each location l of a CDTA by a *difference constraint* $\textit{ineqs}(l)$ which is a positive boolean combination of inequalities between the variables $X(l)$. Intuititively $\textit{ineqs}(l)$ is a condition which is guaranteed to hold between location variables, whenever the automaton passes l. The difference constraints $\{\textit{ineqs}(l) \;:\; l \in L\}$ are the strongest positive combinations of inequalities such that for each $\langle \alpha(\overline{p}), \psi(\overline{x})\rangle \in Dom(\mathcal{T}(l))$ such that $\lambda_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi(\overline{x})\rangle) = \langle \pi, l'\rangle$ we have

$$\pi^{-1}\left(\kappa_{\psi(\overline{x})}\left[\textit{ineqs}(l) \wedge \textit{ineqs}_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi(\overline{x})\rangle)\right]\right) \rightarrow \textit{ineqs}(l')$$

For example, in the automaton in Figure 4.1, location l_1 would have the difference constraint $x_1 \neq x_2$. We say that a CDTA is *determinate* (a DCDTA) if for each location l with difference constraint $\textit{ineqs}(l)$ we have that $\mathcal{T}(l)$ is determinate wrt. any restriction which is consistent with $\textit{ineqs}(l)$.

Consider a constrained word $\langle w, \phi\rangle$ and a DCDTA $\mathcal{A}$.

Define $post_{\mathcal{A}}(\langle w, \phi\rangle)$ as a triple $\langle l, \pi, \zeta\rangle$ where $l \in L$, π is an injective mapping from $X(l)$ to $\Delta_{\langle w, \phi\rangle}$, and ζ is a restriction on $X(l)$. We define $post_{\mathcal{A}}$ inductively, as follows.

- $post_{\mathcal{A}}(\langle \varepsilon, true\rangle) = \langle l_0, \pi_0, true\rangle$ where π_0 is the empty mapping.
- In order to compute $post_{\mathcal{A}}(\langle w, \phi\rangle)$ for some word $w = u\alpha(\overline{p})$, first find the (unique) abstract suffix $\langle \alpha(\overline{p}), \psi(\overline{x})\rangle$ such that $\langle w, \phi\rangle = \langle u, \theta\rangle; \langle \alpha(\overline{p}), \pi(\psi(\overline{x}))\rangle$, where the mapping π is such that $post_{\mathcal{A}}(\langle u, \theta\rangle) = \langle l, \pi, \zeta\rangle$. Then, for any $\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle \in Dom(\mathcal{T}(l))$ such that $\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle \preceq_{\mathcal{T}(l)} \langle \zeta, \langle \alpha(\overline{p}), \psi(\overline{x})\rangle\rangle$, let $\lambda_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle) = \langle \pi', l'\rangle$. Finally, $post_{\mathcal{A}}(\langle w, \phi\rangle) = \langle l', \pi \circ \pi', (\pi')^{-1}(\kappa_{\psi'(\overline{x})}(\zeta \wedge \psi(\overline{x})))\rangle$.

We can now define the function $\lambda_{\mathcal{A}}$ defined by an automaton $\mathcal{A}$ as $\lambda_{\mathcal{A}}(\langle w, \phi\rangle) = \lambda(l)$, where $post_{\mathcal{A}}(\langle w, \phi\rangle) = \langle l, \pi, \zeta\rangle$ for some π and ζ.

Translation from DCDTA to DRA There is a straightforward transformation from DCDTAs to DRAs. From each suffix $\langle \alpha(\overline{p}), \psi(\overline{x})\rangle$ in $Dom(\mathcal{T}(l))$ with $\lambda_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi(\overline{x})\rangle) = \langle \pi, l'\rangle$, we extract a transition $\langle l, \alpha(\overline{p}), g, \pi, l'\rangle$, where

$$g \equiv \psi(\overline{x}) \wedge \textit{ineqs}_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi(\overline{x})\rangle).$$

After this extraction, there may be a potential overlap between some transitions. This is not a severe problem. Because the source CDTA is determinate, overlapping transitions will lead to the same state. However, the resulting DRA will not be deterministic in a strict sense. This can be solved in different ways, e.g., by giving priorities to the transitons.

4.6 A Succinct Nerode Equivalence

We are now able to define a Nerode equivalence, which is more succinct than previously defined equivalences [46, 20].

Let $\lambda : \Phi[(\Sigma_I^P)^*] \mapsto \{+,-\}$ be a language of constrained words. Then any constrained word $\langle u,\theta\rangle$ induces a suffix language, which can be represented by a mapping $\lambda_{[\langle u,\theta\rangle]} : \Phi(\Delta_{\langle u,\theta\rangle})[(\Sigma_I^P)^*] \mapsto \{+,-\}$ from suffixes of $\langle u,\theta\rangle$ to $\{+,-\}$, defined by $\lambda_{[\langle u,\theta\rangle]}(\langle v,\psi\rangle) = \lambda(\langle u,\theta\rangle;\langle v,\psi\rangle)$. Let $\mathcal{T}[\langle u,\theta\rangle]$ be a minimal SCDT for $\lambda_{[\langle u,\theta\rangle]}$. We can then define memorable parameters.

Definition 8 (Memorable)**.** *The* memorable parameters *of a constrained word $\langle u,\theta\rangle \in Dom(\mathcal{T})$, denoted $mem_\lambda(\langle u,\theta\rangle)$, is the set of parameters in $\Delta_{\langle u,\theta\rangle}$ that occur in some constraint ψ in some suffix $\langle v,\psi\rangle \in Dom(\mathcal{T}[\langle u,\theta\rangle])$.* □

Definition 9 (Equivalence)**.** We define the equivalence $\equiv_\lambda$ on $\Phi[(\Sigma_I^P)^*]$ by $\langle u,\theta\rangle \equiv_\lambda \langle u',\theta'\rangle$ if $|\mathsf{mem}_\lambda(\langle u,\theta\rangle)| = |\mathsf{mem}_\lambda(\langle u',\theta'\rangle)|$ and there is a bijection $\gamma : \mathsf{mem}_\lambda(\langle u,\theta\rangle) \mapsto \mathsf{mem}_\lambda(\langle u',\theta'\rangle)$ such that for each (concrete) suffix $\langle v,\psi\rangle$ of $\langle u,\theta\rangle$ we have $\lambda_{[\langle u',\theta'\rangle]}(\langle v,\gamma(\psi)\rangle) = \lambda_{[\langle u,\theta\rangle]}(\langle v,\psi\rangle)$. □

The equivalence $\equiv_\lambda$ is a congruence in the following sense: Let $\langle u,\theta\rangle \equiv_\lambda \langle u',\theta'\rangle$, and let γ be the mapping $\gamma : \mathsf{mem}_\lambda(\langle u,\theta\rangle) \mapsto \mathsf{mem}_\lambda(\langle u',\theta'\rangle)$; this proves the equivalence. Then, for any suffix of $\langle u,\theta\rangle$ of form $\langle\alpha(\overline{p}),\psi\rangle$ we have $\langle u\alpha(\overline{p}),\theta;\psi\rangle \equiv_\lambda \langle u'\alpha(\overline{p}),\theta';\gamma(\psi)\rangle$.

We are now able to state and prove a Myhill-Nerode-like theorem:

Theorem 10 (Myhill-Nerode)**.** *Let $\lambda : \Phi[(\Sigma_I^P)^*] \mapsto \{+,-\}$ be a function on constrained words. Then λ is recognizable by a DCDTA iff $\equiv_\lambda$ has finite index.*

Proof The only-if direction follows from the observation that all prefixes leading to the same location in a DCDTA are equivalent with respect to $\equiv_\lambda$, i.e., that the locations of a DCDTA induce a finer equivalence than $\equiv_\lambda$.

The if-direction follows by constructing a DCDTA from a given $\equiv_\lambda$, as follows.

- *Locations:* The set of locations L is given by the finitely many equivalence classes wrt. to $\equiv_\lambda$. For each equivalence, we choose a representative element. The initial location l_0 is $[\langle\epsilon, true\rangle]_{\equiv_\lambda}$, with the empty word as representative element. The labeling function λ is given by $\lambda[\langle w,\phi\rangle]_{\equiv_\lambda} = \lambda(\langle w,\phi\rangle)$.
- *Variables:* The variables that need to be saved at location $[\langle u,\theta\rangle]_{\equiv_\lambda}$ is the set $\mathsf{mem}_\lambda(\langle u,\theta\rangle)$. It will be convenient to let the variables be just these parameters, so we will confuse parameters and variables in this situation.
- *Transitions:* The transitions from a location $l = [\langle u,\theta\rangle]_{\equiv_\lambda}$ with representative element $\langle u,\theta\rangle$ are given by the SCDT $\mathcal{T}(\langle u,\theta\rangle)$, where $Dom(\mathcal{T}(\langle u,\theta\rangle))$ is the restriction of $\mathcal{T}[\langle u,\theta\rangle]$ to one-symbol suffixes, and $\lambda_{\mathcal{T}(\langle u,\theta\rangle)}(\langle\alpha(\overline{p}),\psi\rangle) = \langle\gamma,[\langle u',\theta'\rangle]_{\equiv_\lambda}\rangle$, where $\langle u',\theta'\rangle$ is the representative element of $[\langle u\alpha(\overline{p}),\theta\wedge\psi\rangle]_{\equiv_\lambda}$, which is established through the mapping $\gamma : \mathsf{mem}_\lambda(\langle u',\theta'\rangle) \mapsto \mathsf{mem}_\lambda(\langle w\alpha(\overline{p}),\phi\wedge\psi\rangle)$.

The constructed DCDTA is well-defined. The set of variables is implicitly defined by the largest set of parameters that has to be remembered in a location. Obviously, the transitions are completely specified and determinate. Since we keep the parameters as variables, no renaming is necessary in the local SCDTs. Thus, the automaton is determinate. The parallel assigments are constructed to transfer and save exactly the memorable variables, while the remapping guarantees that parameters end up in the correct variables. Finally, the accepting locations are the ones corresponding to classes wrt. $\equiv_\lambda$.

We propose the DCDTA that is constructed in the if direction of the above proof as the canconical DRA for $\mathcal{L}$. This automaton is unique up to isomorphism and minimal in a strong sense. The number of locations is determined by the number of classes wrt. $\equiv_\lambda$ and thus minimal; however, minimality only holds with respect with to this particular Myhill-Nerode classification. We will try to strengthen the minimality result in Section 4.8 by comparing our canonical models with DRA in general and with canonical models of a previously formulated Myhill-Nerode-like theorem. Also, obviously, by construction the number of variables is minimal. The minimality of transitions follows directly from using the (minimal) DCDT for $\mathcal{L}$ as a basis for the construction.

4.7 Minimization

In this section, we present a minimization algorithm for DCDTAs. This algorithm will use the pattern of a classical partition-refinement algorithm. It will maintain an equivalence relation between locations, a set of definitely memorable location variables, and a constraint on possible variable mappings between locations.

We need an auxiliary definition. Let $\langle v, \psi(\overline{x})\rangle$ be a suffix, and let $\gamma : \overline{x} \mapsto \overline{x}'$ be a mapping from $\overline{x}$ to some variables $\overline{x}'$. Let ζ_γ be the restriction on $\overline{x}$ obtained as the conjunction of all equalities $x_i = x_j$ such that $\gamma(x_i) = \gamma(x_j)$. Now define $\tilde{\gamma}(\langle v, \psi(\overline{x})\rangle)$ as the $\overline{x}'$-suffix $\langle v, \gamma(\psi'(\overline{x}))\rangle$, where $\psi'(\overline{x})$ is the normalized constraint obtained by removing the equalities between variables in $\zeta \sqcap \psi(\overline{x})$. Intuitively, $\tilde{\gamma}(\langle v, \psi(\overline{x})\rangle)$ is the suffix that results from mapping the variables in $\overline{x}$ according to γ, and performing the appropriate transformations for normalizing the constraint.

Let $\mathcal{A} = (I, L, l_0, \mathcal{T}, \lambda)$ be the DCDTA we want to minimize. The minimization algorithm will maintain the following relations:

- For each location $l \in L$, a set $\mathsf{mem}_k(l) \subseteq X(l)$ of variables that have been found to be memorable.
- For each pair of locations $l, l' \in L$ with $|\mathsf{mem}_k(l)| \geq |\mathsf{mem}_k(l')|$, a set $\Gamma_k(l, l')$ of surjective mappings $\mathsf{mem}_k(l) \mapsto \mathsf{mem}_k(l')$. Intuitively, each mapping $\gamma \in \Gamma_k(l, l')$ indicates that when $\mathcal{A}$ processes suffixes starting from l', then it is possible to process these suffixes also from l, keeping in mind the restriction that is induced by γ.

Initially, we let $\mathsf{mem}_0(l) = \emptyset$ for each l, we let the domain of the SCDT $\mathcal{U}_l$ be $\{\langle \alpha(\overline{p}), true\rangle \ : \ \alpha \in I\}$, and let $\Gamma_0(l, l')$ be the set consisting of the empty mapping if $\lambda(l) = \lambda(l')$ otherwise $\Gamma_0(l, l') = \emptyset$.

Each iteration of the algorithm applies the constraints that these relations must mutually satisfy. At each iteration we perform the following steps:

1. We first compute an SCDT $\mathcal{U}_l$ in each location l, which is a restricted version of $\mathcal{T}(l)$. Namely, we let $Dom(\mathcal{U}_l)$ be the minimal subset of $Dom(\mathcal{T}(l))$ which contains $\langle \alpha(\overline{p}), true\rangle$ for each $\alpha \in I$, and such that whenever $\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle \in Dom(\mathcal{U}_l)$ and $\langle \alpha(\overline{p}), \psi(\overline{x})\rangle \in Dom(\mathcal{T}(l))$ are two suffixes with $\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle \preceq_{\mathcal{U}_l} \langle \alpha(\overline{p}), \psi(\overline{x})\rangle$, and where $\lambda_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi(\overline{x})\rangle) = \langle \pi, m\rangle$ and $\lambda_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle) = \langle \pi', m'\rangle$, then the restriction of $\pi^{-1} \circ \kappa_{\psi(\overline{x})} \circ \pi'$ to $\mathsf{mem}_k(m)$ is in $\Gamma_k(m, m')$ This minimization can be done using a procedure similar to that in the proof of Theorem 4.
2. Next, we extend $\mathsf{mem}_k(l)$ for each l to $\mathsf{mem}_{k+1}(l)$ as follows: Whenever $\lambda_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle) = \langle \pi, m\rangle$ for some $\langle \alpha(\overline{p}), \psi'(\overline{x})\rangle \in Dom(\mathcal{U}_l)$, then $\mathsf{mem}_{k+1}(l)$ must contain each variable in $X(l)$ which is in $\pi(\mathsf{mem}_k(m))$ or occurs in $\psi'(\overline{x})$.
3. Finally, we refine the set $\Gamma_k(l, l')$ to $\Gamma_{k+1}(l, l')$, as follows. The set $\Gamma_{k+1}(l, l')$ should include those surjective mappings $\gamma : \mathsf{mem}_{k+1}(l) \mapsto \mathsf{mem}_{k+1}(l')$ whose restriction to $\mathsf{mem}_k(l)$ is in $\Gamma_k(l, l')$, and satisfy the following condition. For each $\langle \alpha(\overline{p}), \psi'(\overline{x}')\rangle \in Dom(\mathcal{U}_{l'})$ and $\langle \alpha(\overline{p}), \psi(\overline{x})\rangle$ with $\langle \alpha(\overline{p}), \psi'(\overline{x}')\rangle = \tilde{\gamma}(\langle \alpha(\overline{p}), \psi(\overline{x})\rangle)$, let $\langle \alpha(\overline{p}), \psi''(\overline{x})\rangle \in Dom(\mathcal{U}_l)$ be such that $\langle \alpha(\overline{p}), \psi''(\overline{x})\rangle \preceq_{\mathcal{U}_l} \langle \zeta_\gamma, \langle \alpha(\overline{p}), \psi(\overline{x})\rangle\rangle$. Let $\lambda_{\mathcal{T}(l)}(\langle \alpha(\overline{p}), \psi(\overline{x})\rangle) = \langle \pi, m\rangle$, and $\lambda_{\mathcal{T}(l')}(\langle \alpha(\overline{p}), \psi(\overline{x}')\rangle) = \langle \pi', m'\rangle$. Then the mapping γ', such that $\gamma'(y_i) = (\pi')^{-1}(\kappa_{\langle \alpha(\overline{p}), \psi(\overline{x}')\rangle}(\gamma(\pi(y_i))))$ if $\pi(y_i) \in X(l)$ and $\gamma'(y_i) = (\pi')^{-1}(\kappa_{\langle \alpha(\overline{p}), \psi(\overline{x}')\rangle}(\pi(y_i)))$ if $\pi(y_i) \in \overline{p_{\alpha(\overline{p})}}$, should be in $\Gamma_k(m, m')$. Intuitively, this condition checks that if $\mathcal{A}$ processes the restricted suffix $\langle \zeta_\gamma, \langle \alpha(\overline{p}), \psi(\overline{x})\rangle\rangle$ from l' as $\langle \alpha(\overline{p}), \psi'(\overline{x}')\rangle$, then the minimized automaton might do without l', and instead process $\langle \zeta_\gamma, \langle \alpha(\overline{p}), \psi(\overline{x})\rangle\rangle$ from l.

This algorithm terminates, since it is a fixpoint computation over finite domains. After termination, we can obtain the minimized automaton by taking as states the maximal equivalence classes of locations, where l and l' are in the same equivalence class if they have the same number of memorable variables and are related by $\Gamma_k(l, l')$. The set of variables in each location is taken to be $\mathsf{mem}_k(l)$, and the transition relation can be represented by $\mathcal{U}_l$, where l is a representative element of the equivalence class. After that, unreachable equivalence classes may be removed, just as in the previous section, to obtain a minimal set of locations.

4.8 A Hierarchy of Automata Models

An important goal of this chapter is to define a minimal canonic automaton representation of any DRA-recognizable data language by means of a Myhill-Nerode-like theorem. There are already proposals for such a theorem for quasi regular languages [46, 20]. Since it is formulated at the level of concrete data words, the resulting canonical models can be exponentially bigger than the our canonical models are, as we will show.

Let us define an RA that resembles the automata of [20]. An RA is *unique-valued* (called a URA) if the valuation σ in any reachable state $\langle l, \sigma \rangle$ is injective, i.e., two variables are never mapped to the same data value. An RA is *ordered* (called an ORA) if data values are stored only in order of appearance. In an ORA, all assignments in transitions $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle \in T$ thus satisfy the following condition: For any two $x_i, x_j \in X(l')$ with $i < j$ we either have $\pi(x_i) \in X(l)$ and $\pi(x_j) \notin X(l)$, or $\pi(x_i) = p_m$ and $\pi(x_i) = p_n$, or that $\pi(x_i) = x_m$ and $\pi(x_j) = x_n$ respectively imply $m < n$. We will also define an OURA, which is both *ordered* and *unique-valued*.

The automata of [20] correspond to deterministic OURAs (DOURAs). We will use deterministic URAs (DURAs) for an intermediate step, when considering the potential blow-up in the number of locations between DCDTAs and DOURAs.

Proposition 11. *There is a sequence of languages $\mathcal{L}_1, \mathcal{L}_2, \ldots$, such that the number of locations in the minimal DCDTA for $\mathcal{L}_n$ is the same for all n, but the number of locations in the minimal DURA for $\mathcal{L}_n$ is exponential in n.*

Proof Consider the language

$$\mathcal{L}_n = \{a(d_1, \ldots, d_n)b(d'_1, \ldots, d'_n) \ : \ d_j = d'_j \text{ for } 1 \leq j \leq n.\}$$

The minimal DURA for $\mathcal{L}_n$ has an exponential number of locations after the first a-symbol to encode potential equalites between data values: As each unique value can only be stored once in a DURA, one location is needed for every possible partition of $d_1, \ldots, d_n$ wrt. $=$. The minimal DCDTA, on the other hand, has only 4 locations: one initial, one after the first a-symbol, one accepting, and one rejecting sink location.

The second exponential blow-up may occur due to storing values in the order in which they occur.

Proposition 12. *There is a sequence of languages $\mathcal{L}_1, \mathcal{L}_2, \ldots$, such that the number of locations in the minimal DURA for $\mathcal{L}_n$ is constant, but the number of locations in the minimal DOURA for $\mathcal{L}_n$ is exponential in n.*

Proof Let I be the set of symbols a_i and b_i for $i = 1, \ldots, n$, each of which is of arity 1, and c with arity $2n$.

Consider the language

$$\begin{aligned}\mathcal{L}_n = \{&c(d_1, \ldots, d_{2n})a_{l_1}(d_{2n+1}), \ldots, a_{l_m}(d_m)b_k(d_{m+1}) \ : \\ &(1)\ d_i \neq d_j \text{ for } 1 \leq i < j \leq 2n, \\ &(2)\ d_{2n+j} \in \{d_{l_j}, d_{n+l_j}\}, \\ &(3a)\ max\{i | l_i = k\} = c \Rightarrow d_{m+1} = d_{2n+c}, \\ &(3b)\ max\{i | l_i = k\} = \varnothing \Rightarrow d_{m+1} = d_k.\end{aligned}$$

The minimal DURA for $\mathcal{L}_n$ has 4 locations: one initial, one after $c(d_1, \ldots, d_{2n})$ with $2n$ unique variables $x_1, \ldots, x_{2n}$, one accepting and one non-accepting sink. The automaton will immediately move to the sink if (1) is violated during initialization. After the initialization it will loop on any $a_i(d)$ that satisfies (2) and go the sink otherwise: if $d = x_{n+i}$ it will swap the contents of x_i and x_{n+1}. Then, x_i always contains the value d of the last "successful" $a_i(d)$. Finally, it will compare the value d from the final $b_i(d)$ with x_i and accept if $x_i = d$.

The minimal DOURA for $\mathcal{L}_n$, on the other hand, has more than $n!$ locations. Since the DOURA must store all data values in sequence, a control location is needed for every possible reordering of the variables that can be created in the $a_{l_1}(d_{2n+1}), \ldots, a_{l_m}(d_m)$ fragment of accepted words. Otherwise, it will be impossible to correctly judge the final $b_i(d)$-symbol.

Now, let us finally consider the relation of the DRA in general to the canonical DCDTA we defined by means of Theorem 10. DRAs in general may have transition guards implying $(x_i = x_j)$ for two location variables, which leads to different sets of "accessible" transitions in the same location depending on the valuation of variables. DCDTAs on the other hand branch only on equalities between parameters of the currently read symbol and variables. They are "history-independent". We can substantiate this difference by the following proposition.

Proposition 13. *The minimal DRA for any DRA-recognizable language $\mathcal{L}$ has three locations, an initial one, an accepting one and a rejecting one.*

We do not prove this proposition but rather give an idea how such a DRA can be constructed. To encode a DRA with n locations, we introduce $\lceil log_2(n) \rceil$ extra variables for the location information. From the initial location we will create transitions that resemble the original transitions but also encode the destination location in the extra variables. For all other transitions we will extend guards and assignments by elements encoding source and destination location. All transitions will be between the accepting and the rejecting location, depending on acceptance/rejection of the original source and destination.

Such a DRA, however, encodes all state information in a set of variables. This contradicts one of the usual purposes of using automata as a model: encoding control information in locations. On the other hand, it is often not easy do draw an exact line between control and data parts. The Myhill-Nerode theorem we presented in this chapter provides a natural division based on language properties. A "regular skeleton" of the language is represented in the locations of the canonical "history independent" DCDTA, while in the variables the "relevant patterns" in the data values of the accepted data words are encoded.

Actually, "history independence" is the natural criterion that one usually wants to hold for automata. As soon as this is dropped, the location space can be collapsed dramatically.

4.9 Conclusions and Future Work

In this chapter, we present a novel form of register automata, which also has an intuitive and succinct minimal canonical form, which can be derived from a Nerode-like right congruence. We also presented an algorithm for minimization, and proved that our automata can be exponentially more succinct than other proposed canonical forms.

Our immediate plans are to use these results to generalize Angluin-style active learning to data languages over infinite alphabets, which can be used to characterize protocols, services, and interfaces. Another obvious problem is to generalize the canonical model to more expressive signatures with other simple operations on data values, e.g., including comparisons of various forms.

5 Automated Learning of Models with Data

5.1 Introduction

Model-based techniques for verification and testing of software systems have undergone dramatic advances in the last decades [39, 33], and are increasingly being applied in industrial settings (e.g., [61]). They require formal models that specify actual or intended behavior of system components. Ideally, models should be developed during specification and design; this, however, typically requires significant manual effort, implying that in practice models are often not available, or become outdated as the system evolves.

There is an increasing interest in automated techniques that support the construction of models. Static program analysis techniques can be used to construct models from source code (e.g., [18, 56, 103]), but are often of limited use due to the unavailability of source code in library modules, third-party components, etc. Increased attention is devoted to black-box techniques for constructing models from observations of their external behavior. Indeed, models generated by black-box techniques have been used, e.g., for regression testing [54, 62], integration testing [53], to support program evolution [44], and to analyze security protocols [100].

Most black-box teachniques for model generation fall in one of two categories. One category generates finite-state automata models that represent sequences of interactions between a component and its environment [54, 62, 15, 100], using regular inference (aka automata learning) techniques (e.g., [16, 93, 71]. Another category generates invariants over state variables [44] or exchanged data values. For many applications in testing and verification, however, it is important to learn models that capture combined behavior of control and data. Parameters such as sequence numbers, identifiers, etc. have a significant impact on control flow in typical protocols. For instance, a valid sequence number or session identifier has a very different influence on continued behavior than an invalid one. This influence of data on control flow is taken into account by model-based test generation tools, such as ConformiQ Qtronic [61].

In this chapter, we present an algorithm for generating models that describe both finite-state control behavior, how a component manipulates data, and the interaction between data and control. The generated models are finite-state machines extended with data from an unbounded domain. Data values can appear as parameters in interactions at the component interface, stored in state variables, and tested against previously stored data values. Within CONNECT, being able to learn rich models of networked components is one of the major challenges of WP4.

In this particular setting, we restrict the allowed operation on parameter values to equality tests; the set of operations will be extended in future work. One motivation is to handle parameters that, e.g., are user names, passwords, identifiers of connections, sessions, etc. similar to, and slightly more expressive than, the class of “data-independent” systems, which was the subject of some of the first works on model checking of infinite-state systems [107, 65].

Other algorithms have been presented for generating similar models that combine control and data. Our work distinguishes itself from those by (i) being fully automated and implemented, and (ii) completely capturing the interaction between data and control. Some previous work combines automata learning and monitoring of data values in a rather shallow way: first a finite-state control structure is created using automata learning, and then constraints on exchanged data values are generated for each interaction in this control structure [74, 75, 82]. This approach cannot capture influences of data on control, e.g., as in the above example of sequence numbers. Our previous work [24] (as well as that by Sakamoto [95]) presents an approach for learning a similar class of automata as in this chapter, which creates an very large intermediate finite transition graph obtained by instantiating each interaction primitive with a sufficiently large number of separate data values; this approach appears impractical for implementation.

While in earlier works we used rudimentary abstraction techniques to arrive at feasible models without focusing explicitly the combination of inference and abstraction [54], recently we developed approaches, inspired by predicate abstraction, which combine classical automata learning with abstraction [8, 60]; however, in one approach the abstraction must be created manually, which may not always be trivial, while in the other relations between parameters will not be made explicit. In [10] the relations between

system under test, abstraction, and inferred model are investigated formally.

The algorithm we present here is an extension of L^* to the class of register automata described above. We have developed a novel representation of register automata, which enjoys precisely the properties of automata that are exploited by L^*. This allows transferring L^* to the new setting with only minor modifications. The parallel with L^* also means that the algorithm maintains strong ties with results in automata theory: learned models are canonical, and they are minimal in a strong sense. In the remainder of the introduction, we will attempt to explain how our algorithm parallels L^*, and highlight the novel innovations that we bring.

Let us begin with an intuitive summary of the popular L^* algorithm for active learning. Active learning attempts to construct an automaton that recognizes a given unknown target language $\mathcal{L}$ via observation/-experimentation. The L^* algorithm [16] uses two kinds of queries:

- *Membership queries*, asking whether a certain word lies in $\mathcal{L}$, and
- *Equivalence queries*, asking whether a hypothesized automaton $\mathcal{H}$ is correct, i.e., recognizes $\mathcal{L}$. If $\mathcal{H}$ is incorrect, a word which separates $\mathcal{L}$ from $\mathcal{H}$ is returned.

L^* aims to construct the minimal DFA $\mathcal{M}_\mathcal{L}$ that recognizes $\mathcal{L}$, by exploiting the Myhill-Nerode characterization of $\mathcal{M}_\mathcal{L}$ [57]: any word u represents a state in $\mathcal{M}_\mathcal{L}$, namely that which is reached from the initial state after reading u. Two words u and u' represent the same state if $uv \in \mathcal{L} \Leftrightarrow u'v \in \mathcal{L}$ for any word v, i.e., if they cannot be distinguished by any suffix.

The L^* algorithm systematically asks membership queries with the goal to collect enough information to characterize $\mathcal{M}_\mathcal{L}$. The algorithm maintains two increasing sets of words:

- a set *prefixes* of words which represent an expanding subset of the states of $\mathcal{M}_\mathcal{L}$,
- a set *suffixes* of words which is used to distinguish states.

During the algorithm, the set *prefixes* may contain representatives of only some of the states of $\mathcal{M}_\mathcal{L}$, and the set *suffixes* may be too small to distinguish representatives that lead to different states in $\mathcal{M}_\mathcal{L}$. However, upon termination, the set *prefixes* contains representatives for each state in $\mathcal{M}_\mathcal{L}$, and *suffixes* contains enough suffixes to distinguish representatives of different states.

The typical behavior of L^* is to start by asking a sequence of membership queries according to certain rules. This process converges when a hypothesized DFA $\mathcal{H}$ can be built from the obtained answers, upon which L^* makes an equivalence query to find out whether $\mathcal{H}$ is equivalent to $\mathcal{L}$. If the result is successful, the algorithm terminates, otherwise the returned counterexample is used as a driver for additional membership queries, until converging at a new hypothesized DFA, etc.

Theoretically, this Myhill/Nerode-based pattern can be directly applied to data languages as well, but at the cost of non-termination: in general, data languages will give rise to infinitely many equivalence classes, and therefore to automata with infinitely many states. As an example, consider a language $\mathcal{L}_{login}$ of successful user authentications.

$$\mathcal{L}_{login} = \{\mathsf{reg}(d_1, d_2)\mathsf{login}(d_3, d_4) \mid d_1 = d_3 \wedge d_2 = d_4\}.$$

where the parameters d_1, d_2, d_3, d_4 may assume integer values (say). Then any combination of values d_1, d_2 is inequivalent to any other combination of values d_1, d_2.

A way out of this dilemma is using a symbolic representation of words. We can define a *constrained word* as a pair $\langle w, \phi \rangle$ consisting of a word w, in which data parameters are "free" to be constrained by the constraint ϕ. An example could be $\langle \mathsf{reg}(p_1, p_2)\mathsf{login}(p_3, p_4) \ , \ p_1 = p_3 \wedge p_2 = p_4 \rangle$. A straightforward application of the Myhill-Nerode pattern using constrained words would reduce the number of equivalence classes to a finite number, but we can do better. As an example, the constrained word $\langle \mathsf{reg}(p_1, p_2) \ , \ p_1 = p_2 \rangle$ would be inequivalent to the constrained word $\langle \mathsf{reg}(p_1, p_2) \ , \ p_1 \neq p_2 \rangle$, although the equality $p_1 = p_2$ is irrelevant for the language.

In our work, we have developed a more succinct representation of data languages, which represents a language by a (typically small) number of "essential" constrained words. Intuitively, a constrained word is

essential if each of its atomic predicates is necessary for determining whether the constrained word will be accepted or rejected. Thus, a data language can be represented by a uniquely defined set of "essential" constrained words, and a mapping from these to $\{+, -\}$. This mapping can then be naturally extended to the set of all data words: A data word inherits the mapping to $\{+, -\}$ from the essential constrained word whose symbolic constraint best matches it.

Locations of our constructed automaton will be represented by "essential" constrained words (playing the role of prefixes), and that two constrained words represent the same location if they cannot be distinguished by any "essential" suffix, just as in L^*. We need only provide one more construction in order to generate minimal automata, which are uniquely defined for any language.

This construction arises from the fact that suffixes in our setup contains references to parameters of a prefix. For instance, in the above example $\langle \mathsf{login}(p_3, p_4)\ ,\ p_1 = p_3 \wedge p_2 = p_4 \rangle$ is a suffix which contains the "free" parameters p_1 and p_2. When matching prefixes and suffixes, there are in general several ways to match the parameters of a prefix with the free parameters of a suffix. We say that two essential words are equivalent wrt. $\mathcal{L}$ if there exists a re-mapping between formal parameters of the two words such that they cannot be distinguished by any suffix. The necessity of re-mapping reflects the fact that it is not possible to control the use of the registers during the learning process in a way that automatically guarantees a perfect match. Instead, we need to "re-shuffle" the register contents of prefixes such that different prefixes leading to the same location behave identical according to the considered suffixes. This necessity is the cause of the main complication when extending L^* to data languages.

In summary, in order to generalize active automata learning to data languages, we need

- to move from the treatment of concrete words to a more symbolic treatment of constrained words, which requires
- an enhanced mechanism for establishing adequate connections between prefixes and suffixes.

Although the general pattern of Angluin's L^* is maintained, these requirements lead to changes in almost every phase of the learning procedure. Particularly involved is the required treatment of counter examples for maintaining the invariant that only essential words will be encountered.

Related Work. Regular inference techniques have been used for several tasks in verification and test generation, e.g., to create models of environment constraints with respect to which a component should be verified [40], for regression testing to create a specification and test suite [54, 62], to perform model checking without access to source code or formal models [52, 88], for program analysis [15], and for formal specification and verification [40].

Extensions to cover models with data values have been considered in several works. In [23], we showed how guards on boolean parameters can be refined lazily. Groz, Li, and Shahbaz [73, 99, 53] extend regular inference to Mealy machines with data values, for use in integration testing. They use only a finite set of the data values in the obtained model, and do not infer internal state variables. Lorenzoli, Mariani, and Pezzé infer models of software components in which a finite, data-insensitive, control structure capturing the possible sequences of method invocations, is inferred by passive learning (by an extension of the k-tails algorithm), and a "pattern-inference" approach using Daikon [34] is used to infer guards and relations on parameters in such methods [75, 82]. In contrast, we use an active learning approach and also capture the interdependence between allowed data values and past invocation history.

Techniques for learning automata which can handle data from an unbounded domain and test for equality between past and subseqeunt data values have been proposed by Sakamoto [95] and in our earlier work [24], using an approach which is different from this chapter. Standard regular inference is used to infer a finite automaton, which represents the behavior on a sufficiently large finite domain of data values. These works suffer from the generation of large finite automata: there are no clear complexity bounds in the papers. Another drawback is that they are specialized for the case where data values are from the same domain and equality is the only way to compare them. The approach in this chapter adapts to other predicates on data values more easily.

Aarts, Jonsson, and Uijen [8] have presented a general framework for inferring models of systems with data parameters: a manually constructed abstraction transforms the system into a finite-state one, to

which standard regular inference can be applied. This method relies on manual support to construct the particular abstraction, and is not automated as in this chapter. A particular form of infinite-state models is that of timed automata, for which for which specialized algorithms have been developed [50, 49, 106].

Organization. We give basic definitions of our automaton model in the next section. In Section 5.3 we will introduce the constrained word representation of data languages resulting in a succinct Nerode-equivalence. We present our inference algorithm in Section 5.4. Experimental results are presented in Section 5.5, before we conclude in Section 5.6.

5.2 Data Languages and Automata

We assume a domain D of data values and a set I of *action types*. Each action type α has a certain *arity*, which determines how many parameters it takes from the domain D. We also assume a set of binary predicate symbols. In this chapter, we will only use equality $=$.[1]

A *data symbol* is a term of the form $\alpha(d_1, \ldots, d_n)$, where α is an action type with arity n, and $d_1, \ldots, d_n$ are data values in D. We define Σ_I^D as the set of data symbols of the form $\alpha(d_1, \ldots, d_n)$, where $\alpha \in I$. A *data word* is a sequence of data symbols. A *data language* $\mathcal{L}$ is a set of data words, which is closed under permutations on D. In other words, a data language must treat all data values symmetrically. This is a natural assumption, e.g., in (web) services that expose the same behavior to every user, or network protocols that pass along session identifiers.

We will next present an automaton model that recognizes data languages. Assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, and a finite set of *variables* (or registers), ranged over by $x_1, x_2, \ldots$. A *parameterized symbol* is a term of form $\alpha(p_1, \ldots, p_n)$, consisting of an action type α and formal parameters $p_1, \ldots, p_n$ (respecting the arity of α). A *guard* is a conjunction of atomic predicates of form $z_i = z_j$ or $z_i \neq z_j$, where each of z_i and z_j is either a formal parameter or a variable. We write $\overline{p}$ for $p_1, \ldots, p_n$, $\overline{d}$ for $d_1, \ldots, d_n$, and $\overline{x}$ for $x_1, \ldots, x_k$.

Definition 14. A *Register Automaton* (RA) is a tuple $\mathcal{A} = (I, L, l_0, T, \lambda)$, where

- I is a finite set of action types,
- L is a finite set of *locations*; with each location l, we associate a tuple $X(l)$ of variables,
- $l_0 \in L$ is the *initial location*, with $X(l_0)$ being the empty tuple,
- T is a finite set of *transitions*, each of which is of form $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle$, where l is a *source location*, $\alpha(\overline{p})$ is a parameterized symbol, g is a guard over $\overline{p}$ and $X(l)$, π (the *assignment*) is a mapping from $X(l')$ to $X(l) \cup \overline{p}$, and l' is a *target location*, and
- $\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject). □

We write $l \xrightarrow{\alpha(\overline{p}):g/\pi} l'$ to denote that $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle \in T$. The RA should be *completely specified*, meaning for any location l and input action α, the disjunction of all guards g in transitions of form $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle$ in T is equivalent to $true$. The RA is *deterministic* (called a DRA) if for any location l and input action α, the conjunction $g_1 \wedge g_2$ is unsatisfiable whenever $\langle l, \alpha(\overline{p}), g_1, \pi_1, l_1' \rangle$ and $\langle l, \alpha(\overline{p}), g_2, \pi_2, l_2' \rangle$ are different transitions from the same location with the same input action in T.

For a tuple X of variables, a X-valuation is an assignment $\sigma^X : X \mapsto \mathcal{D}$ of data values to the variables in X. Valuations are extended to predicates and guards in the natural way.

Let us define the semantics of an RA $\mathcal{A}$. A *state* of a RA $\mathcal{A} = (I, L, l_0, T, \lambda)$ is a pair $\langle l, \sigma^{X(l)} \rangle$ where $l \in L$ and $\sigma^{X(l)}$ is a $X(l)$-valuation. The *initial state* is $\langle l_0, \sigma^{\emptyset} \rangle$. A *run* of $\mathcal{A}$ over a data word $\alpha_1(\overline{d}_1) \cdots \alpha_k(\overline{d}_k)$ is a sequence of states $\langle l_0, \sigma^{\emptyset} \rangle \ \langle l_1, \sigma_1^{X(l_1)} \rangle \cdots \langle l_, \sigma_k^{X(l_k)} \rangle$ such that $\langle l_0, \sigma^{\emptyset} \rangle$ is the initial state, and for each i with $1 \leq i \leq k$ there is a transition from l_{i-1} to l_i, $\langle l_{i-1}, \alpha_i(\overline{p}_i), g_i, \pi_i, l_i \rangle \in T$ such that $\sigma_{i-i}^{X(l_{i-1})}(g[\overline{d}_i/\overline{p}_i])$ is true, and such that $\sigma_i^{X(l_i)}(x_j) = \sigma_{i-1}^{X(l_{i-1})}(\pi(x_j))$ if $\pi(x_j) \in X(l)$ and $\sigma_i^{X(l_{i-1})}(x_j) = d_l$ if $\pi(x_j) = p_l$ for p_l in $\overline{p}$. Such a run is *accepting* if $\lambda(l_k) = +$, otherwise it is *rejecting*. A data word is *accepted* by $\mathcal{A}$ if $\mathcal{A}$ has an accepting run over it. The language recognized by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$ is the set of data words that it accepts. A state $\langle l, \sigma \rangle$ is *reachable* if $\langle l_0, \sigma^{\emptyset} \rangle \xrightarrow{w} \langle l, \sigma \rangle$ for some data word w.

[1] In future work, we plan to cover simple extensions, such as a total order $<$, or membership $\in$.

5.3 A Succinct Nerode Equivalence

In this section we will present a succinct Nerode equivalence for data languages, which is the conceptual "backbone" of our learning algorithm. Additionally, this allows us to introduce the technical notation needed for the formulation of the learning algorithm, e.g., splitting words into prefixes and suffixes.

Constrained Words In general, there are many (deterministic) register automata that accept a given data language. Central to the L^* algorithm is to have canonical automaton representation of languages. In this section, we present a canonical automaton model, which is introduced in Chapter 4. Its basic idea is to minimize the set of atomic predicates that are used in transitions from each location. To define it, we need to look a bit more carefully at how we can organize the set of atomic predicates in transitions.

Recall that a parameterized symbol is of form $\alpha(p_1, \ldots, p_n)$ where $p_1, \ldots, p_n$ are formal parameters. Let Σ_I^P denote the set of parameterized symbols $\alpha(\overline{p})$ with $\alpha \in I$. A *parameterized word* (or just *word*) is a sequence of parameterized symbols in which all formal parameters are distinct; we use $\overline{p_w}$ to denote the ordered sequence of formal parameters in w. A *constraint* is a conjunction of atomic predicates over formal parameters (i.e., of form $p_i = p_j$); note that a constraint does not contain negated predicates. A *constrained word* is a pair $\langle w, \phi \rangle$ consisting of a parameterized word w and a constraint ϕ over the formal parameters $\overline{p_w}$ of w. For a set $W \subseteq (\Sigma_I^P)^*$ of parameterized words, let $\Phi[W]$ denote the set of constrained words $\langle w, \phi \rangle$ with $w \in W$.

Let $\overline{p_w}$ be the ordered sequence $p_1, \ldots, p_n$. We require that constraints be written on a normal form, namely as an ordered conjunction $p_{i_1} = p_{j_1} \wedge p_{i_2} = p_{j_2} \wedge \cdots \wedge p_{i_k} = p_{j_k}$, where

1. each atomic predicate $p_{i_l} = p_{j_l}$ is an ordered pair with $i_l < j_l$, and
2. whenever $p_i = p_j$ appears before $p_{i'} = p_{j'}$, then $i \neq i'$ and $j < j'$.

Each constraint ϕ then has a unique normal form. By the associativity of conjunction, we can write it as an ordered list $p_{i_1} = p_{j_1} \wedge p_{i_2} = p_{j_2}, \cdots \wedge p_{i_k} = p_{j_k}$ where $j_1 < \cdots < j_k$ and where all left hand sides of equalities are distinct. In the following, whenever we write a constraint as $\theta \wedge \psi$, we assume θ and ψ are normalized constraints such that their concatenation $\theta \wedge \psi$ is immediately on normal form. For the general case, we let $\phi \sqcap \phi'$ denote the normalized constraint equivalent to the conjunction of ϕ and ϕ'. Let $\phi \sqsubseteq \phi'$ denote that ϕ' implies ϕ. We use $true$ to denote the empty constraint.

Obviously there is a function from data words to constrained words. We can then represent a data language $\mathcal{L}$ by the unique mapping $\lambda : \Phi[(\Sigma_I^P)^*] \mapsto \{+, -\}$ from the set of constrained words to $\{+, -\}$ in the obvious way.

We proceed to define the *potential* of a constrained word $\langle u, \theta \rangle$, denoted $\Delta_{\langle u,\theta \rangle}$, as the set of formal parameters in u that do not occur as the left argument of any atomic predicate in θ. These can be constrained in atomic predicates when $\langle u, \theta \rangle$ is extended by a suffix.

Let $\langle u, \theta \rangle$ be a constrained word. A *concrete* $\langle u, \theta \rangle$*-suffix* (or just concrete suffix) is a tuple $\langle v, \psi \rangle$, where v is a parameterized word, and ψ is a normalized constraint of atomic predicates of form $p_i = p_j$, where $p_i \in \Delta_{\langle u,\theta \rangle} \cup \overline{p_v}$ and $p_j \in \overline{p_v}$. To the parameters of ψ that are not in $\overline{p_v}$, we refer to as the *free* parameters of $\langle v, \psi \rangle$. For a prefix $\langle u, \theta \rangle$ and a concrete suffix $\langle v, \psi \rangle$, let $\langle u, \theta \rangle; \langle v, \psi \rangle$ denote $\langle uv, \theta \wedge \psi \rangle$.

Recall that $x_1, x_2, \ldots x_k$ range over variables. An *abstract suffix* is a pair $\langle v, \psi(\overline{x}) \rangle$, where v is a parameterized word, and $\psi(\overline{x})$ is a normalized conjunction of atomic predicates $z = p_j$, where $z \in \overline{x} \cup \overline{p_v}$ and $p_j \in \overline{p_v}$. By normalized, we mean as before that right-hand sides are ordered and left-hand sides are distinct. We can easily construct an abstract suffix from a concrete suffix by replacing its free parameters by variables.

Essential Words We have developed a succinct representation of data languages. A data language is defined by a (typically small) set of *essential constrained words* together with a mapping from this set to $\{+, -\}$. Intuitively, a constrained word is essential if each of its atomic predicates is essential for determining whether the constrained word will be accepted or rejected. Note that the set of essential constrained words is different for different data languages, of course.

Let us now define the notion of "essential". Consider a mapping λ from constrained words to $\{+,-\}$. The set of essential constrained words is defined inductively, as follows:

- $\langle w, true\rangle$ is essential for any parameterized word w.
- If $\langle u,\theta\rangle$ is essential, then $\langle uv,\theta\rangle$ is essential for any continuation v of u.
- If $\langle u,\theta\rangle$ is essential and ϕ is a constraint over parameters of u with atomic predicates $z = p_i$ such that $j < i$ for any formal parameter p_j that occurs in θ, then $\langle u, \theta\wedge\phi\rangle$ is essential iff there is a $\langle u,\theta\rangle$-suffix $\langle v,\psi\rangle$ such that $\lambda(\langle uv, \theta\wedge\phi'\rangle \sqcap \psi\rangle) \neq \lambda(\langle uv, (\theta\wedge\phi)\sqcap\psi\rangle)$ for any $\phi' \sqsubseteq \phi$.

A *constraint decision tree* (CDT) $\mathcal{T}$ from W to $\{+,-\}$ is a pair $\langle Dom(\mathcal{T}), \lambda_{\mathcal{T}}\rangle$ where $Dom(\mathcal{T}) \subseteq \Phi[W]$ (the set of "essential" constrained words) is a prefix-closed (wrt. W) set of constrained words which contains $\langle w, true\rangle$ for each $w \in W$, and $\lambda_{\mathcal{T}} : Dom(\mathcal{T}) \mapsto R$ is a mapping from $Dom(\mathcal{T})$ to R.

Conversely, any CDT defines a data language: intuitively, a data word inherits the mapping to $\{+,-\}$ from the essential constrained word whose symbolic constraint best matches it.

Nerode equivalence We can now define a Nerode-type congruence on essential constrained words. Assume a data language $\mathcal{L}$ and an essential constrained word $\langle u,\theta\rangle$. Then a concrete $\langle u,\theta\rangle$-suffix $\langle v,\psi\rangle$ is *essential after* $\langle u,\theta\rangle$ if $\langle u,\theta\rangle;\langle v,\psi\rangle$ is essential.

The above is a very concise definition of essential suffixes. We could also provide a more concrete, equivalent, definition of the set of suffixes that are essential after $\langle u,\theta\rangle$ in the same style as the definition of essential constrained words. In this chapter, we omit the details.

For a CDT $\langle Dom(\mathcal{T}), \lambda_{\mathcal{T}}\rangle$ and a constrained word $\langle u,\theta\rangle \in Dom(\mathcal{T})$, let the $\langle u,\theta\rangle$-residual of $\mathcal{T}$, denoted by $\langle u,\theta\rangle^{-1}\mathcal{T}$, be a pair $\langle Dom(\mathcal{T})_{\langle u,\theta\rangle}, \lambda_{\mathcal{T}}^{\langle u,\theta\rangle}\rangle$, where $Dom(\mathcal{T})(\langle u,\theta\rangle)$ is the set of essential suffixes after $\langle u,\theta\rangle$, and $\lambda_{\mathcal{T}}^{\langle u,\theta\rangle}(\langle v,\psi\rangle) = \lambda_{\mathcal{T}}(\langle u,\theta\rangle;\langle v,\psi\rangle)$.

The learning algorithm presented in the next section will use partial residuals, which we will refer to as closures, to store information. For a prefix $\langle u,\theta\rangle$, let a $\langle u,\theta\rangle$-*closure* be a partial mapping from concrete $\langle u,\theta\rangle$-suffixes to $\{+,-\}$. We use $\mathcal{C}$ to range over closures, and sometimes write $\mathcal{C}(\overline{p})$ and use the term $\overline{p}$-*closure*, to denote that the free parameters in suffixes are in $\overline{p}$. Let *Suff* be a closed set of abstract suffixes. Then a $\langle u,\theta\rangle$-closure $\mathcal{C}$ *covers Suff* if $\mathcal{C}(\langle v,\psi\rangle) = \lambda^{\mathcal{L}}_{\langle u,\theta\rangle}(\langle v,\psi\rangle)$ for all concretizations of suffixes in *Suff* over $\Delta_{\langle u,\theta\rangle}$.

Residuals induce an equivalence on essential prefixes. In our learning algorithm we will use closures to incrementally approximate this equivalence on a set of essential prefixes. We prove in Chapter 4 that this equivalence has finite index precisely when the original data language can be recognized by a finite DRA. Here, we will use a relaxed version of this equivalence that only holds for essential words.

Definition 15 (Equivalence of essential words)**.** *Two essential parameterized words* $\langle u,\theta\rangle$ *and* $\langle u',\theta'\rangle$ *are equivalent wrt.* $\equiv_{\mathcal{L}}$ iff

$$\exists\gamma\ \forall\langle v,\psi\rangle\ .\ \langle u,\theta\rangle;\langle v,\psi\rangle \in \mathcal{L} \Leftrightarrow \langle u',\theta'\rangle;\gamma(\langle v,\psi\rangle) \in \mathcal{L},$$

Intuitively, two essential words are equivalent wrt. $\mathcal{L}$ if there exists a re-mapping between formal parameters of the two words such that the residuals become identical under this mapping.

5.4 Inference of Data Automata

Our algorithm for inference of data languages is similar to the L^* algorithm [16]. It tries to infer an uknown data language $\mathcal{L}$, of which it initially knows only the set of input actions, by asking two kinds of queries.

- A *membership query* consists in asking if a constrained word $\langle w,\phi\rangle$ is in $\mathcal{L}$.
- An *equivalence query* consists in asking whether a hypothesized DRA $\mathcal{H}$ is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. The query is answered by yes if $\mathcal{H}$ is correct, otherwise by a *counterexample*, which is a constrained word $\langle w,\phi\rangle$ that w.l.o.g. is in $\mathcal{L}$ but is rejected by $\mathcal{H}$.

Basing on the Nerode-like equivalence for data languages, our algorithm will perform the exact same steps as a learning algorithm for regular languages. We will also maintain three sets of words in order to approximate the equivalence relation on a set of essential words. We will discuss the data structures and the first step of closing the observations is Section 5.4.1. We will in particular have to spend some effort on maintaining the memorable parameters of the prefixes and on determining the bijections γ between prefixes.

As in the classical case, we will formulate hypothesis automata (DRA) once we have completed the observations. We will focus on how to generate guards and parallel assignments for the transitions from the observations. We will discuss this step in Section 5.4.2.

Finally, we will have to analyze counterexamples. We will follow the pattern from [93]. Only, for constrained words some extra effort will be needed in order to identify the new prefix or suffix, which are guaranteed to lead to a better approximation of $\equiv_{\mathcal{L}}$. In particular, we will show in Section 5.4.3, how to integrate splitting of the counterexample and weakening its constraint locally in a way that allows us to exactly identify the new prefix or suffix.

5.4.1 Observation Tables

Following L^* [16, 93], we use a table-like data structure for organizing results of queries, for direct the generation of future queries, and for generating hypothesized automata. In L^*, the table is a mapping from pairs of prefixes and suffixes to $\{+, -\}$. We will use a similar data structure over prefixes and abstract suffixes.

An observation table $\mathcal{T}$ is characterized by a prefix-closed set of prefixes *Pref*($\mathcal{T}$) and a set of abstract suffixes *Suff*($\mathcal{T}$). The set *Pref*($\mathcal{T}$) is the union of a prefixed-closed subset $Sp(\mathcal{T})$ of *short prefixes*, and a set of one-symbol extensions of the prefixes in $Sp(\mathcal{T})$ which contains at least the prefix $\langle u, \theta\rangle; \langle \alpha(\overline{p}), true\rangle$ for each $\langle u, \theta\rangle \in Sp(\mathcal{T})$. We will refer to the set of one-symbol extensions by $Lp(\mathcal{T})$.

The table $\mathcal{T}$ maps each prefix $\langle u, \theta\rangle \in Dom(\mathcal{T})$ to a $\langle u, \theta\rangle$-closure $\mathcal{T}(\langle u, \theta\rangle)$, which covers *Suff*($\mathcal{T}$). The closure $\mathcal{T}(\langle u, \theta\rangle)$ is, of course, constructed by asking membership queries for all concatenations of form $\langle u, \theta\rangle; \langle v, \psi\rangle$ for some $\langle u, \theta\rangle$-instance $\langle v, \psi\rangle$ of some abstract suffix in *Suff*($\mathcal{T}$).

From an observation table, we will construct an automaton by letting prefixes that are mapped to equivalent closures go to the same location. The equivalence between closures may involve renaming of free parameters, since different parameters in different closures may have similar roles. If $\overline{p}$ and $\overline{p}'$ are two sets of parameters, then a *remapping* from $\overline{p}$ to $\overline{p}'$ is an bijection γ from $\overline{p}$ to $\overline{p}'$.

For a $\overline{p}$-closure $\mathcal{C}(\overline{p})$, and a remapping γ from $\overline{p}$ to $\overline{p}'$, we define $\gamma(\mathcal{C}(\overline{p}))$ as the $\overline{p}'$-closure $\mathcal{C}'$. We derive $\mathcal{C}'$ from $\mathcal{C}$ by replacing all occurences of parameters p from $\overline{p}$ in suffixes from $Dom(\mathcal{C})$ by $\gamma(p)$.

We say that two closures $\mathcal{C}(\overline{p})$ and $\mathcal{C}'(\overline{p}')$ are *equivalent* if $\overline{p}$ and $\overline{p}'$ contain the same number of variables, and there is a bijective remapping γ from $\overline{p}$ to $\overline{p}'$, such that $\gamma(\mathcal{C}(\overline{p})) = \mathcal{C}'(\overline{p}')$. In this case we write $\mathcal{C}(\overline{p}) \simeq_\gamma \mathcal{C}'(\overline{p}')$; we write $\mathcal{C}(\overline{p}) \not\simeq \mathcal{C}'(\overline{p}')$ if there is no such remapping.

We can now describe the actual learning algorithm. Initially, we will have $Sp(\mathcal{T}) = \{\langle \varepsilon, true\rangle\}$, and $Lp(\mathcal{T}) = \{\langle \alpha, true\rangle \ : \ \alpha \in I\}$ as the prefixes of the table, and *Suff*($\mathcal{T}$) $= \{\langle \varepsilon, true\rangle\}$ as suffixes.

To construct a well-defined hypothesis from the observations, certain properties have to hold on the table. The learning algorithm will thus continue by checking and establishing the following three properties on the observation table.

Location Closedness The set of short prefixes will later be used to represent the states of a hypothesis automaton. We need to ensure that all one-letter extensions, which will be used to generate the transitions, end in a state of the hypothesis. We say that $\mathcal{T}$ is closed if for any prefix $\langle u, \theta\rangle \in Lp(\mathcal{T})$ there is a prefix $\langle u', \theta'\rangle \in Sp(\mathcal{T})$, and some remapping γ such that $\mathcal{T}(\langle u, \theta\rangle) \simeq_\gamma \mathcal{T}(\langle u', \theta'\rangle)$.

If there exists $\langle u, \theta\rangle \in Lp(\mathcal{T})$ such that for all $\langle u', \theta'\rangle \in Sp(\mathcal{T})$ we have $\mathcal{T}(\langle u, \theta\rangle) \not\simeq \mathcal{T}(\langle u', \theta'\rangle)$, we will move $\langle u, \theta\rangle$ from $Lp(\mathcal{T})$ to $Sp(\mathcal{T})$. Intuitively, this will eventually close the set of of states in the hypothesis under the transition relation. We will extend $Lp(\mathcal{T})$ accordingly by the one-letter extensions of $\langle u, \theta\rangle$.

Variable Consistency To be able to construct a hypothesis from the table, we need information about which parameters are to be stored into state varaibles along transitions. The parameters that need to be stored at the state representing $\langle u, \theta \rangle \in Sp(\mathcal{T})$ are simply the free parameters of the $\langle u, \theta \rangle$-closure, which by construction can be mapped to the free parameters of any $\langle u', \theta' \rangle \in Sp(\mathcal{T})$ where $\mathcal{T}(\langle u, \theta \rangle) \simeq_\gamma \mathcal{T}(\langle u', \theta' \rangle)$.

Now, the parameters to be stored have to be consistent for all states. If at some state a parameter has to be remembered that does not belong to the most recent input, then this parameter has to be stored as a variable in a preceding state. As suffixes indicate that a parameter has to be remebered at a certain state, we will use suffixes to "propagate" information about parameters to be stored.

Let $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle \in \textit{Pref}(\mathcal{T})$ a prefix for which we know from $\langle v, \psi \rangle \in \mathcal{T}(\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle)$ that a parameter p from the potential of $\langle u, \theta \rangle$ has to be stored as state variables, i.e., $p \in \psi \cap \Delta_{\langle u, \theta \rangle}$.

Then, let p not be in the free parameters of $\mathcal{T}(\langle u, \theta \rangle)$. In order to propagate the requirement to store p, we will extend $\textit{Suff}(\mathcal{T})$ by $\langle \alpha v, (\phi \sqcap \psi)(\overline{x}) \rangle$, such that $\langle v, \psi \rangle$ was a $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle$-instance of $\langle v, \psi(\overline{x}) \rangle$. Please observe that this will maintain the suffix-closedness of $\textit{Suff}(\mathcal{T})$.

Transition Closedness In order to be able to construct deterministic automata, we need to ensure that we can derive disjoint transitions from the prefixes of $\mathcal{T}$. We will later "determinize" guards constructed from constraints of prefixes by adding negations of more specific constraints and incomparable constraints. In order to do this successfully, we need to close the set of transitions under meet. For every two words of form $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle$ and $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi' \rangle$ in $\textit{Pref}(\mathcal{T})$, where $\phi \not\sqsubseteq \phi'$ and $\phi' \not\sqsubseteq \phi$, we add $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \sqcap \phi' \rangle$ to $Lp(\mathcal{T})$.

In each of the above steps, the observation table is extended and the closures will have to be extended accordingly by means of membership queries. Once the observations are complete and all conditions are met, we can formulate a hypothesis automaton from the table.

5.4.2 Constructing Hypotheses

Let $\tau_{\langle u, \theta \rangle}$ be a mapping from a set of variables, ranged over by $x_1, \ldots x_k$, to the free parameters of the $\langle u, \theta \rangle$-closure. Intuitively, τ will tell us, which parameter to store in which variable when reaching a location from different transitions. We will have ${\tau_{\langle u, \theta \rangle}}^{-1} \circ \tau_{\langle u', \theta' \rangle} = \gamma$ for two words reaching the same state, and γ being the remapping which both closures become equal. (We assume that γ always maps the parameters of a prefix in $Lp(\mathcal{T})$ to the parameters of a short prefix.)

Our algorithm will maintain the invariant that $\mathcal{T}(\langle u, \theta \rangle) \not\simeq \mathcal{T}(\langle u', \theta' \rangle)$ for any two short prefixes $\langle u, \theta \rangle$ and $\langle u', \theta' \rangle$. We can thus construct the hypothesis $\mathcal{H}_\mathcal{T} = (I, L, l_0, T, \lambda)$:

- There is exactly one location $l_{\langle u, \theta \rangle}$ in the set of locations L for every $\langle u, \theta \rangle \in Sp(\mathcal{T})$.
- The initial location l_0 is the location corresponding to the empty word $\langle \epsilon, true \rangle$.
- The function λ can be constructed from the observations as $\langle \epsilon, true \rangle$ is in the set of suffixes. We define $\lambda(l_{\langle u, \theta \rangle}) = \mathcal{T}(\langle u, \theta \rangle)(\langle \epsilon, true \rangle)$.
- The variables that need to be stored at some location $l_{\langle u, \theta \rangle}$ are determined using $\tau_{\langle u, \theta \rangle}$.
- We use all one-symbol extensions of short prefixes to generate the transitions. From $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle$ in $Lp(\mathcal{T})$ we extract a transition $\langle l_{\langle u, \theta \rangle}, \alpha(\overline{p}), g, \pi, l_{\langle u', \theta' \rangle} \rangle$, where $\mathcal{T}(\langle u', \theta' \rangle) \simeq_\gamma \mathcal{T}(\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle$ for $\langle u', \theta' \rangle \in Sp(\mathcal{T})$.

  Let $g = {\tau_{\langle u, \theta \rangle}}^{-1}(\phi)$, i.e., the references to parameters in u are replaced by references to variables. Let the references of parameters in $\alpha(\overline{p})$ be not affected by this operation.

  Then, let $\rho = \tau_{\langle u', \theta' \rangle} \circ \gamma^{-1}$, and let $\pi(x_i) = \rho(x_i)$ if $\rho(x_i)$ is a parameter of $\langle \alpha(\overline{p}), \phi \rangle$ and $\pi(x_i) = \rho \circ {\tau_{\langle u, \theta \rangle}}^{-1}(x_i)$ otherwise.

The DRA constructed this way is not deterministic yet, because transition guards are not disjoint. In order to make the guards disjoint, we will add constraints to them.

Let $\sqsubseteq$ be extended to guards in the obvious way and let $P = \{\langle \alpha(\overline{p}), \phi\rangle \ : \ \langle u, \theta\rangle; \langle \alpha(\overline{p}), \phi\rangle \in \mathit{Pref}(\mathcal{T})\}$ be the set of one-letter extensions for some short prefix $\langle u, \theta\rangle$. For every $\langle \alpha(\overline{p}), \phi\rangle \in P$ there is a corresponding transition and guard in $\mathcal{H}_\mathcal{T}$, which we refer to by $g_{\langle \alpha(\overline{p}), \phi\rangle} \in P_{\langle \alpha(\overline{p}), \phi\rangle}$. Then, let $g' \prec_P g$ for $g, g' \in P_{\langle \alpha(\overline{p}), \phi\rangle}$ denote that g' is a strongest (wrt. $\sqsubseteq$) guard in $P_{\langle \alpha(\overline{p}), \phi\rangle}$, and $g' \sqsubseteq g$ but $g \not\sqsubseteq g'$. We extend any guard $g \in P_{\langle \alpha(\overline{p}), \phi\rangle}$ to $g \wedge \neg g'$ of all $g' \in P_{\langle \alpha(\overline{p}), \phi\rangle}$, where $g \prec_P g'$, i.e., all the guards that are "slightly stronger" than g. We further extend g to $g \wedge \neg g''$ for $g'' \in P_{\langle \alpha(\overline{p}), \phi\rangle}$ with $g \neq g''$, where $g'' \not\prec_P g$, and $g \not\prec_P g''$, and g'' is minimal wrt. $\sqsubseteq$, i.e., there is no $g' \in P_{\langle \alpha(\overline{p}), \phi\rangle}$ with $g \neq g'$, where $g' \not\prec_P g$, and $g \not\prec_P g'$, and $g' \prec_P g''$.

Intuitively, the different guards for transitions over the same parameterized symbol describe several hierarchies of implications between constraints from more specific ones down to the most general one ($true$). By extending each guard by the negations of the slightly stronger guards and the weakest incomparable guards, we guarantee that every guard is disjoint with other guards implying or "overlapping" it. Enforcing *transition closedness* on the table will ensure that we have exactly one most specific element per hierarchy of transitions and thus that the result is a completely specified, deterministic register automaton.

The constructed DRA is now well-defined: The set of locations is defined by the short prefixes of $\mathcal{T}$, and the prefix $\langle \varepsilon, true\rangle$ for l_0 will always be in $Sp(\mathcal{T})$. The sets of variables are defined by the free parameters in the closures of the short prefixes. Due to *location closedness*, the hypothesis is closed under the transition relation. The transitions are disjoint and the DRA is completely specified. *Variable consistency* guarantees that the parallel assignments are well-defined. Finally, the labelling λ reflects the classfification of all prefixes in the observation table wrt. the language to be learned.

5.4.3 Handling Counterexamples

Once we have generated a hypothesis, we can use an equivalence query in order to test if the language accepted by the current hypothesis coincides with the target language. In case the languages do not coincide yet, the equivalence query will return with a counterexample, i.e, a word that w.l.o.g. is in the language $\mathcal{L}$ but is rejected by the current hypothesis $\mathcal{H}$.

As pointed out in Section 5.1, we will work on essential constrained words, i.e., constrained words where every predicate of the constraint is essential to being a counterexample. Let us first describe how to transform a data word (or a corresponding constrained word) that is a counterexample into an essential constrained word that contains only relevant constraints.

Essential constrained words We will use the data word $\alpha_1(d_1)\alpha_2(d_1)\alpha_3(d_1)\ldots\alpha_m(d_1)$ as an example. We process such a word from left to right and peform the following step for every data value.

If the data value at the current position does not occur earlier in the word, we just proceed with the next data value. If the data does occur earlier, we test if this equality is essential, i.e., if the word stops being a counterexample when removing it. In our example we would have to test the relevance of the equality of d_1 in $\alpha_2(d_1)$ and $\alpha_1(d_1)$.

Since, however, the currently tested data value can also occur at later positions of the word, removing the equality is not trivial. Both data values may be required to be equal to any of the later occurences. In our example, this could apply to the last d_1. To test this, we will replace the data value at the current position (d_1 of $\alpha_2(d_1)$ in the example) by a new data value d_n that does not yet occur in the counterexample. Now, we test all possible partitions of the subsequent d_1 into d_1 and d_n. If we find a combination that still makes a counterexample, the original equality was not essential. We can proceed with the new word and the next data value. Otherwise, we have to proceed with the old counterexample and the next data value. Upon termination only relevant equalities between data values remain, the corresponding constrained word is essential.

Essential counterexamples An essential counterexample is a constrained word $\langle w', \phi'\rangle$, which w.l.o.g. is in the target language but rejected by the current hypothesis. (We have not formally defined the notion of runs for constrained words but leave this as an exercise to the reader). To exract a new prefix or a new suffix from a counterexample, we will step-wisely transform $\langle w', \phi'\rangle$ to its "access sequence" $\lfloor\langle w', \phi'\rangle\rfloor_\mathcal{H}$

on the hypothesis, i.e., the word that leads to the same location in $\mathcal{H}$, and test when exactly it stops being a counterexample. Since $\lambda_{\mathcal{H}}(\langle w', \phi' \rangle) = \lambda_{\mathcal{H}}(\lfloor \langle w', \phi' \rangle \rfloor_{\mathcal{H}})$, and thus $\lfloor \langle w', \phi' \rangle \rfloor_{\mathcal{H}} \notin \mathcal{L}$, it is guaranteed that we will be successful.

We will split a counterexample into three parts: a prefix u, a symbol α, and a suffix v. We denote this by:

$$\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle; \langle v, \psi \rangle,$$

where we let θ contain all $(p_i = p_j)$ with $p_j \in \overline{p}_u$, and ϕ will contain all $(p_i = p_j)$ where $p_j \in \overline{p}_\alpha$. Finally, ψ will consist of all the $(p_i = p_j)$ where $p_j \in \overline{p}_v$.

Now, we need to get some information from the hypothesis. Intuitively, from a run of $\mathcal{A}$ for $\langle w', \phi' \rangle$, we can determine "slightly weaker" versions of ϕ and ψ. Let therefore $\tau_{\langle u,\theta \rangle}$ be a mapping from the variables of the location reached by $\langle u, \theta \rangle$ to the correspondig (stored) parameters of $\langle u, \theta \rangle$. (A straight-forward inductive defintion of τ can be provided by defining runs over constrained words.) Let additionally $t \in T$ be the transition that corresponds to the step of $\mathcal{H}$ on $\langle \alpha(\overline{p}), \phi \rangle$.

We will relax $\langle w', \phi' \rangle$ according to t such that the constraint ϕ in the resulting word corresponds exactly to the guard g of t. Then, we will relax ψ so it contains only predicates that are compatible with the parallel assignment π of t, i.e., such that it only refers to parameters (of the prefix) that by $\mathcal{H}$ are stored in a variable after processing t.

1. Let $\lfloor \phi \rfloor_g \sqsubseteq \phi$ be the constraint that for every predicate $(p_i = p_j) \in g$ contains the pair $(p_i = p_j)$, and for every predciate $(p_i = x_i) \in g$ contains the pair $(\tau_{\langle u,\theta \rangle}(x_i) = p_i)$.
2. Let $\lfloor \psi \rfloor_\pi \sqsubseteq \psi$ be the constraint that contains all $(p_i = p_j) \in \psi$, for which either $p_i \in \overline{p}_v$ or p_i is in the image of $\tau_{\langle u,\theta \rangle; \langle \alpha(\overline{p}), \phi \rangle}$.

Beginning with index 0, we generate the following three words for every index i of the counterexample and test if each of this words is in $\mathcal{L}$ using membership queries.

$$\begin{aligned}
c_{i,1} &= \langle u, \theta \rangle\ ; \ \langle \alpha(\overline{p}), \phi \rangle\ ; \ \langle v, \psi \rangle \\
c_{i,2} &= \langle u, \theta \rangle\ ; \ \langle \alpha(\overline{p}), \lfloor \phi \rfloor_g \rangle\ ; \ \langle v, \psi \rangle \\
c_{i,3} &= \langle u, \theta \rangle\ ; \ \langle \alpha(\overline{p}), \lfloor \phi \rfloor_g \rangle\ ; \ \langle v, \lfloor \psi \rfloor_\pi \rangle \\
c_{i+1,1} = c_{i,4} &= \lfloor \langle u, \theta \rangle\ ; \ \langle \alpha(\overline{p}), \lfloor \phi \rfloor_g \rangle \rfloor_{\mathcal{H}}\ ; \ \gamma(\langle v, \lfloor \psi \rfloor_\pi \rangle)
\end{aligned}$$

There, γ is provided from the observations.

If now $c_{i,1} \in \mathcal{L}$ but $c_{i,2} \notin \mathcal{L}$, the counterexample provides evidence that $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle \not\equiv_{\mathcal{L}} \langle u, \theta \rangle; \langle \alpha(\overline{p}), \lfloor \phi \rfloor_g \rangle$. Since the counterexample was essential, it will be sufficient to add the word $\langle u, \theta \rangle; \langle \alpha(\overline{p}), \phi \rangle$, to the set $Lp(\mathcal{T})$ in order to represent this extra transition.

If $c_{i,2} \in \mathcal{L}$ but $c_{i,3} \notin \mathcal{L}$, the counterexample shows that along the transition t in $\mathcal{H}$ relevant variables are not kept. To represent the relevance of this variables in the observations, it will be sufficient to add the abstract suffix $\langle v, \psi(\overline{x}) \rangle$ and all its suffixes to the table.

Finally, if $c_{i,3} \in \mathcal{L}$ but $c_{i,4} \notin \mathcal{L}$, the suffix $\langle v, \lfloor \psi \rfloor_\pi \rangle$ proves the remapping γ incorrect. Adding the corresponding abstract suffix $\langle v, \psi(\overline{x}) \rangle$ and all its suffixes to the table will lead to a refined remapping or, in case no valid refinement is possible, to a new location.

In case, all three words are in $\mathcal{L}$, we continue with the next index, using the word

$$\lfloor \langle u, \theta \rangle; \langle \alpha(\overline{p}), \lfloor \phi \rfloor_g \rangle \rfloor_{\mathcal{H}}; \gamma(\langle v, \lfloor \psi \rfloor_\pi \rangle)$$

as a basis, and $\lfloor \langle u, \theta \rangle; \langle \alpha(\overline{p}), \lfloor \phi \rfloor_g \rangle \rfloor_{\mathcal{H}}$ as new $\langle u, \theta \rangle$.

Correctness follows trivially: For a counterexample $\langle w', \phi' \rangle$ we start the above process using the empty prefix for u. The process ultimately leads to $\lfloor \langle w', \phi' \rangle \rfloor_{\mathcal{H}}$. As pointed out, at some particular point during the transformation acceptance will turn into rejection.

Adding all suffixes of an abstract suffix to the table will ensure that every hypothesis accepts a different language than the previous one. The argument behind this is rather technical. Intuitively, if we add only one suffix we might end up with a hypothesis in the next round that has one state more but the new state accepts the same residual as some other state. Suffix-completeness of the set of abstract suffixes will guarantee that all new states are distinguishable.

5.4.4 Correctness and Complexity

The procedure discussed in the previous sections results in an algorithm following the L^* pattern: The algorithm will perform in rounds, and each round ends with an equivalence query. The steps in a single round guarantee that a well-defined hypothesis automaton can be generated from the observations. Processing of counterexamples will lead to a refined observation table from which a new hypothesis can be constructed that differs from the last hypothesis in one of four possible ways: it can have more transitions, more locations, more variables, or a different remapping between transitions reaching a location. For any regular data language $\mathcal{L}$ the algorithm will terminate with a minimal deterministic RA for $\mathcal{L}$. Minimality of states and variables follows by construction. Minimality of the number of transitions follows from the minimality of closures.

Let then v be the number of variables a system uses, i.e., the maximum number of variables that are stored in a single location. By n we denote the number of locations, by t the number of transitions, and by m the length of the longest counterexample. Let p be the number of parameters of the symbol with the most parameters; then, by construction, the potential of a prefix in *Pref*($\mathcal{T}$) is bounded by np.

Let us first discuss the maximal number of rounds. Every counterexample can either deliver a new prefix or new abstract suffixes. A maximum of nv suffixes can indicate that variables are not stored. A maximum of n^2 suffixes may be needed to reduce the number of possible remappings (or to distinguish locations). [2] This results in less than $t + n^2 + nv$ rounds. We will assume that v is in $O(n)$ and simplify to $O(t + n^2)$.

The size of the observation table will be bounded by the number of prefixes and the number of suffixes. The number of prefixes will be the number of transitions in the automaton plus one for the empty prefix. This is in $O(t)$. The number of suffixes will be bounded by the number of rounds in which suffixes are added to the table and the maximum length of counterexamples. Suffixes from *variables consistency* will not add to this estimate. With every suffix less than m suffixes will be added to the table. In total, the number of (abstract) suffixes will be in $O(n^2m)$.

All operations on the table, except finding remappings, are polynomial in the size of the table. Finding remappings is exponential in v. There are at most $v!$ possible remappings between two closures.

The number of membership queries is bounded by the size of the table, by the number of membership queries needed to produce one closure, and by the number of queries needed to process counterexamples. An abstract suffix can have at most v free variables, which can be instanciated by less than np parameters in the potential of a word in less than $(np)^v$ combinations. The number of membership queries needed to construct all closures is then in $O(t \cdot n^2m \cdot (np)^v)$.

The analysis of a counterexample will cost less than $mp2^{mp}$ queries. At each iteration of making a counterexample essential, there are at most 2^{mp} many partitons of less than mp subsequent equal data values.

Altogether, the number of membership queries will be in $O(tm(np)^v + (t + n^2) \cdot mp2^{mp})$. The maximal number of equivalence will be in $O(t + n^2)$, corresponding to the number of rounds.

Obviously, many optimizations are possible. E.g., there is much information in the closures about potential transitions that is not exploited currently. Pre-initializing *Suff*($\mathcal{T}$) in a fashion resembling the approach in [102], could help reducing the number of equivalence queries.

On the other hand, some costs are unlikely to be reduced. The DRA we construct are exponentially more concise than the canocial DRA for data languages from [20]. The exponential savings result from introducing the constrained word interpretation of data language. The exponential costs for the derivation of essential counterexamples and the construction of closures indicate that both types of models contain the same information.

Table 5.1: Observation Table (only showing a subset of all prefixes)

	γ	$\langle\varepsilon, true\rangle$	$\binom{x_1}{x_2}\ in(x_1,x_2)$		$\binom{x_1}{x_2}\ out()\ in(x_1,x_2)$	
$\langle\varepsilon, true\rangle$ (l_0)	$-$	$-$	$true$	$-$	$true$	$-$
$reg(p_1,p_2)$ (l_1)	$v_1 \leftarrow p_1$ $v_2 \leftarrow p_2$	$-$	$true$ $\binom{p_1 \leftarrow x_1}{p_2 \leftarrow x_2}$	$-$ $+$	$true$ $\binom{p_1 \leftarrow x_1}{p_2 \leftarrow x_2}$	$-$ $+$
$reg(p_1,p_2)\ in(p_3,p_4)^{\phi_1}$ (l_2)	$v_1 \leftarrow p_3$ $v_2 \leftarrow p_4$	$+$	$true$	$+$	$true$ $\binom{p_3 \leftarrow x_1}{p_4 \leftarrow x_2}$	$-$ $+$
$reg(p_1,p_2)\ in(p_3,p_4)\ out()^{\phi_1}$	$p_3 \leftarrow p_3$ $p_4 \leftarrow p_4$	$-$	$true$ $\binom{p_3 \leftarrow x_1}{p_4 \leftarrow x_2}$	$-$ $+$	$true$ $\binom{p_3 \leftarrow x_1}{p_4 \leftarrow x_2}$	$-$ $+$

5.4.5 Example Run of the Algorithm

In this section, we provide an example run of our algorithm. Our example is a language, which represents successful user authentications to a fictitious system. The language consists of sequences of symbols of form reg(d_1, d_2) and login(d_1, d_2). Intuitively, the first parameter d_1 represents a user name, while the second parameter d_2 represents a password. We extend the example by the possibility to logout. The language $\mathcal{L}_{login}$ of successful user authentications is the set of words

$$\mathcal{L}_{login} = \{\text{reg}(d_1, d_2)(I)^*\text{login}(d_3, d_4) \mid d_1 = d_3 \wedge d_2 = d_4\},$$

where $I = \{\text{reg}(p_1, p_2), \text{login}(p_1, p_1), \text{logout}\}$, which we will abbreviate by $I = \{\text{reg}(p_1, p_2), \text{in}(p_1, p_1), \text{out}\}$ sometimes.

We start by asking membership queries for $\langle\varepsilon, true\rangle$ and all the one-symbol continuations of $\langle\varepsilon, true\rangle$. The table is now closed and consistent, because all symbols lead to the nonaccepting state. We construct a hypothesis (one non-accepting state) and get a counterexample, which we assume is $reg(p_1, p_2)\ in(p_3, p_4)^{\phi_1}$, where $\phi_1 = \{p_1 = p_3 \wedge p_2 = p_4\}$.

We now treat the counterexample as described in Section 5.4.3. This will finally lead to replacing its first half by the corresponding short prefix $\langle\varepsilon, true\rangle$. Executing this on the hypothesis, however, still leads to the nonaccepting state, so we add the abstract suffix $\binom{x_1}{x_2}\ in(x_1, x_2)$ to the table.

We fill out the column, and discover for the prefix $reg(p_1, p_2)$ that it deviates from the default case only when mapping x_1 to p_1 and x_2 to p_2 so we add this special case to the table cell. We also notice that this mapping is not reflected in any prefix yet, so through location closedness we make $reg(p_1, p_2)$ a short prefix, and add all its one-symbol continuations to the table.

The table is now closed. We generate a hypothesis with two non-accepting states from this table (one corresponding to $\langle\varepsilon, true\rangle$ and one corresponding to $reg(p_1, p_2)$). We will get the same counterexample as in the first round. Analyzing it will lead to adding $\langle reg(p_1, p_2)\ in(p_3, p_4), p_1 = p_3 \wedge p_2 = p_4\rangle$ as a prefix to the table. Completing the table, we move it to the set of short prefixes and add all its one-symbol continuations.

Through variables consistency, we then notice that for $reg(p_1, p_2)\ in(p_3, p_4)\ out()$ some of the parameters from the short prefix $reg(p_1, p_2)\ in(p_3, p_4)$ have to be stored as state variables. We thus add the abstract suffix $\binom{x_1}{x_2}\ out()\ in(x_1, x_2)$ to the table and fill the column. Now the table is closed and consistent again, so we construct the final model, shown in Figure 5.1.

5.5 Experimental Results

We have implemented the outlined algorithm prototypically on top of LearnLib [92, 72], which provides a rich infrastructure for developing learning algorithms and conducting case studies. The implementation

[2] n suffixes will suffice per location because they suffice to distinguish n locations. Once we make all locations "visible" in a single closure, only actual symmetries between parameters and corresponding remappings will remain.

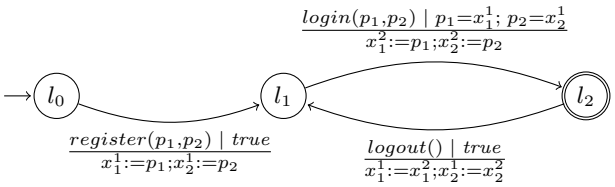

Figure 5.1: Final hypothesis (partly defined)

comprises all the modules discussed in previous sections: data structures to organize closures and the observation table, generation of hypothesis models, and handling counterexamples.

As black-box systems, we used three small data structures and a fragment of the XMPP (Extensible Messaging and Presence Protocol) protocol [104]. XMPP is a protocol that is widely used in instant messaging, extensions also cover streaming multimedia content. We used primitives of XMPP to register an account, log in with an account, change the password, and delete the account. For the data structures, Java implementations served as SUT, while for the XMPP fragment, we instrumented an actual XMPP server.

When learning the systems, we did not perform membership queries using constrained words directly, but instead used the corresponding concrete data words. Essential counterexamples were provided manually. In future applications, we envision using model-based testing methods as a realization of equivalence queries.

The three small data structures we used all have the purpose of storing a limited number of objects, so they expose two parameterized actions: $put(p)$ and a $get(p)$. We implemented one data structure with stack-like semantics, one with queue-like semantics, and one with bag-like semantics. *Putting* was possible as long as the size of the data structure was not exceeded. *Getting* would fail if the corresponding semantics were violated, e.g., on the stack $get(p)$ had to request the p of the most recent $put(p)$. All three implementations would allow storing the same object multiple times. The size of the data structures was limited to three objects, but we also conducted an extra series of experiments on the stack with up to four storable objects. We implemented the data structures to accept valid sequences of input and otherwise go to a nonaccepting sink state.

When doing the XMPP case study, we had to consider the fact that the our learning algorithm was formulated for data languages. For this reason, we defined an accepted trace as one where the user was authenticated after executing it.

Table 5.2: Experimental Results

Example	# Loc.	# Trans.	MQs	EQs	Time [s]
Stack (1)	3	7	35	2	0
Stack (2)	4	10	135	4	1
Stack (3)	5	13	554	6	4
Stack (4)	6	16	2,998	8	13
Queue	5	13	554	6	5
Bag	5	16	134	7	3
XMPP	3	16	355	2	143

Table 5.2 shows the key figures of the experimental results. The data structures each have one location per number of stored objects and one error sink. The XMPP model has three states, one initial, one registered and one logged in. The resulting models (for the data structures for three objects and the XMPP fragment) are partly shown in Figure 5.2. For the sake of readability, the error sinks and some reflexive transitions are omitted in the figure. For that same reason, we also transformed the direct result of our algorithm into a model that uses global variables.

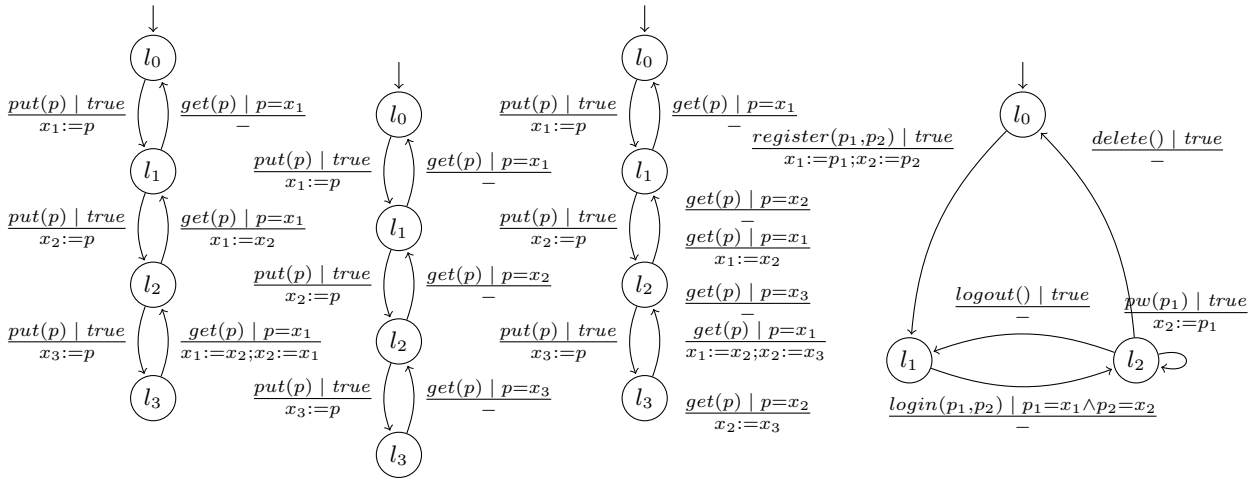

Figure 5.2: Partial models for Queue, Stack, Bag, and a fragment of XMPP (from left to right)

Looking at the table, we observe that especially the running times and number of queries seem promising. This makes us confident that the algorithm will also work well on larger systems, as long as the number of variables is relatively small compared to the number of locations. From the series of experiments with the stack it becomes apparent that the number of membership queries can grow exponentially with the number of variables, though.

However, if we compare the costs of applying our learning algorithm with those of a classical L^*, this algorithm would quickly use more membership queries than ours. We see that the smallest sensible data domain for inferring a data structure for k objects would require $2k$ alphabet actions. In the resulting model, every combination of k stored objects would then be encoded in the set of locations (roughly $\sum_k i^k$). Then, assuming n to be the number of locations and a the number of actions, the minimal size of an observation table would be $na \cdot log_2(n)$, giving us a rather conservative estimate for the number of membership queries needed by a classic learning algorithm. For a stack accepting two, three, or four objects, this estimate would already result in 72, 1,404, and 24,820 queries, respectively.

Comparing the results for the data structures, it may not be immediately intuitive that the bag implementation would have more transitions, require more counterexamples, and yet require substantially fewer membership queries than the other two implementations. There is, however, a logical explanation for this. In a bag, stored variables can be used symmetrically; it does not matter which of a number of identical values is removed. In contrast, in the other data structures, variables have to be used in relation to their order of appearance. The discovery of such asymmetric behavior is realized in our algorithm using additional suffixes that disprove re-mappings (cf. Section 5.4.3). We can also see it in the case of the data structures, where the observation tables for queue and stack of size three contain four abstract suffixes, while the one for the bag has only two.

In conclusion, we would like to point out that the resulting models are visual proof of one main benefit of our method. When using a classical learning algorithm, one could easily obtain a model with more than 340 locations for a stack of size four. In contrast, our models are concise, and the relevant relations between data values are encoded into variables and formal parameters in a way that makes them directly understandable.

5.6 Conclusions and Future Work

In this chapter, we have presented an extension of active learning to data languages, which allows it to produce models that cover the influence of data parameters on the control flow. Our extension includes

a systematic treatment of constraints that relate different occurrences of data values in sequences of interactions, and of state variables that must be introduced in state machine models to enforce such constraints. We have demonstrated the capabilities of our new method in a number of small case studies. Next on our agenda is to extend the new method to a broader set of relations between data parameters and to systems with output.

Also, with the RERS (regular extrapolation of reactive systems) initiative [59] we are trying to establish a community of researchers and practitioners in the field interested in the further development and application of regular inference techniques.

6 Dynamic Refinement of Abstractions

Abstraction is the key when learning behavioral models of realistic systems, but also the cause of a major problem: the introduction of non-determinism. In this chapter, we introduce a method for refining a given abstraction to automatically regain a deterministic behavior on-the-fly during the learning process. Thus the control over abstraction becomes part of the learning process, with the effect that detected non-determinism does not lead to failure, but to a dynamic alphabet abstraction refinement.

In the context of CONNECT this method will be valuable when it comes to automatic generation of test-drivers (cf. Chapter 2). A major step in the automation of test-driver generation will be the automatic production of abstractions. It is highly unlikely that an approach can be found that always immediately generates a correct and meaningful abstraction. Automated alphabet abstraction refinement will enable the automatic correction of "good" abstractions (in a sense to be defined more precisely below) rather than trying to produce perfect abstractions. The remainder of this chapter is published as [60].

In D4.1 we presented a number of case studies in which automata learning was applied in different scenarios relevant to CONNECT. All these scenarios had one thing in common: the learned systems were 'wrapped' in a kind of test harness, which in addition to providing access to the system to be learned also took care of the abstraction inherent in the definition of the considered behavioral perspective, like hiding certain parameters, abstracting time to causality, or treating certain resources (e.g., phones) symbolically. This meant that the learning algorithm was not confronted with the real system, but only with its wrapper-based abstraction, whose adequacy therefore was critical to the success of the whole learning enterprise.

We present this abstraction in Fig. 6.1. Here, Σ^I denotes an input alphabet and Σ^O an output alphabet. By Σ_C we denote an alphabet at the concrete level, while Σ_A refers to an abstract (symbolic) alphabet. As the employed active learning techniques assume a deterministic behavior, this meant in particular that this abstraction (from Σ_C to Σ_A) had to impose a deterministic behavior on the concrete system. This is a very strong requirement when dealing with black box systems, which led to many manual modifications of the wrapper in the course of a single learning experiment, each one requiring a complete restart of the learning process, using the refined alphabet.

In this chapter we propose a method to automatically refine a given abstraction until a level is reached where this abstraction imposes a deterministic behavior on the concrete system. Like automata learning itself, this method is in general neither sound nor complete, but it also enjoys similar convergence properties even for infinite systems (in fact, both the alphabet and the state set of the concrete systems may by infinite) as long as the concrete system itself behaves deterministically. Key to this method is the switch from the learning scenario shown in the left of Fig. 6.1 to the one in the right, which allows the learning algorithm (L) to control the abstraction. Technically this is achieved by a change of perspective: Rather than working at the abstract level, the learner is sitting now at the concrete level in order to observe the concrete system behavior for a set of representatives of the equivalence classes imposed by the abstraction. Thus abstraction is no longer a 'filter' between the concrete system and the learning algorithm,

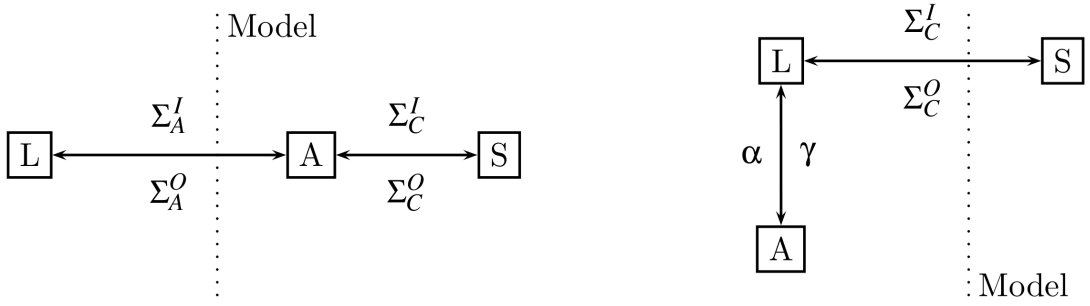

Figure 6.1: traditional use of abstraction (left) and abstraction as part of the learning process (right)

but rather a teacher, helping the learner to choose adequate representative tests. This learner is able to automatically resolve *controllable* non-determinism, i.e. non-determinism which is due to the imposed abstraction. Thus the control over abstraction becomes part of the learning process, with the effect that detected non-determinism does not lead to failure, but to a dynamic refinement of the abstraction.

In Section 6.1 we introduce the theoretical framework we use for alphabet abstraction refinement and show the existence of a coarsest deterministic alphabet abstraction. Section 6.2 presents the integration of the proposed abstraction refinement into existing active learning algorithms, while Section 6.3 discusses an illustrating example scenario, before we conclude in the final section.

6.1 Alphabet Abstraction Refinement

Mealy automata have turned out to be a very good automata model for learning in practice. However for the ease of exposition, we will introduce our new method for *alphabet abstraction refinement* (AAR) first for the structurally simpler setting of deterministic automata. Please note that we do not require the finiteness of the alphabet or the set of states. They are not necessary for the correctness of the method, but only for the convergence of the corresponding algorithm. The subsequent generalization to the setting of (countable) Mealy automata is then straightforward.

In order to emphasize that the systems we are aiming at do not need to be finite state themselves, we directly introduce the following notion of countable automata. Of course, automata which we produce will continue to be finite state: they are finite state views/approximations of potentially infinite state systems. This is why we prefer to call this learning process regular extrapolation (see also RERS challenge [36]).

Definition 16. *A deterministic countable automaton (DCA) is a tuple* $Sys = \langle Q, q_0, \Sigma, \delta, F \rangle$ *where*

- Q *is a countable nonempty set of states,*
- $q_0 \in Q$ *is the initial state,*
- Σ *is a countable alphabet,*
- $\delta : Q \times \Sigma \to Q$ *is the transition function, and*
- $F \subseteq Q$ *is the set of accepting states.*

Intuitively, a DCA evolves through states $q \in Q$*, and whenever one applies an input symbol (or action)* $a \in \Sigma$*, the machine moves to a new state according to* $\delta(q, a)$*. A word* $w \in \Sigma^*$ *is accepted by the DCA if and only if the DCA reaches an accepting state* $q_i \in F$ *after processing the word starting from its initial state. We write* $q \xrightarrow{a} q'$ *to denote that on input symbol a the DCA moves from state* q *to state* q'*. The transition function* $\delta : Q \times \Sigma \to Q$ *can be extended to* $\delta' : Q \times \Sigma^* \to Q$ *such that for all states* $q, q' \in Q$ *letters* $a \in \Sigma$ *and words* $w \in \Sigma^*$ *the following holds:* $\delta'(q, \epsilon) = q$*, and* $\delta'(q, aw) = \delta'(\delta(q, a), w)$*.*

The following section introduces the notion of *determinism preserving abstraction* (DPA) and states that any abstraction has a unique greatest DPA. Our learning algorithm, subsequently presented in Section 6.2, resolves detected non-determinism of a given abstraction by (optimal) refinement. Rather than failing in response to non-determinism, the algorithm automatically also 'learns' the corresponding greatest DPA.

6.1.1 Determinism Preserving Abstraction

Assume an unknown DCA Sys together with a finite abstraction given in terms of a pair of adjoint functions (α, γ), i.e., $\alpha : \Sigma_C \to \Sigma_A$, where sets annotated with C denote sets of (potentially infinite) concrete alphabet symbols and sets annotated with A denote sets of finite abstract input symbols. The set of abstract input symbols Σ_A will evolve throughout the learning process according to the refinement in response to detected non-determinism. The concretization function $\gamma : \Sigma_A \to \Sigma_C$ is required to satisfy that $\gamma \circ \alpha$ is the identity: i.e., each $\gamma(a)$ has to be an element of $\alpha^{-1}(a)$.

Like for verification, when learning behavioral models of real systems, finding the right level of abstraction is essential. It is necessary to make the learning problem tractable, and it allows one to automatically arrive at tailored views focusing on the parts of interest. Let us therefore assume that Σ_A is a finite abstraction of the concrete alphabet Σ_C, identifying what we would like to distinguish of the behavior of Sys, and $\alpha : \Sigma_C \rightarrow \Sigma_A$ is the corresponding (abstraction) function. By α-equivalence we denote the equivalence relation $\equiv_\alpha$ over Σ_C^* induced by α, i.e.:

$$\forall v, w \in \Sigma^* \,.\, v \equiv_\alpha w \;\Leftrightarrow\; |v| = |w| \wedge\; \forall 1 \leq i \leq |v|.\; \alpha(v_i) = \alpha(w_i)$$

Typical prerequisite of active learning techniques is that there exists a deterministic acceptor (the *membership oracle*) for individual behaviors (words). Unfortunately, even if Sys itself would be such a deterministic acceptor[1], its abstraction to observations in Σ_A^* is in general not deterministic, i.e., there exist words $w_1, w_2 \in \Sigma_C^*$ with

- Sys accepts w_1 but rejects w_2 and
- w_1 and w_2 are α-equivalent.

This has the fatal effect that typical active learning solutions would simply fail in their attempt to learn the behavior at this level of abstraction. For the rest of this section we will show that there exists a 'best' or coarsest intermediate abstraction $\alpha_{c,o} : \Sigma_C \rightarrow \Sigma_{c,o}$ which refines α in a deterministic fashion, i.e.:

$$\forall w_1, w_2 \in \Sigma^* \,.\, w_1 \equiv_\alpha w_2 \implies (w_1 \in L(Sys) = w_2 \in L(Sys))$$

Abstractions like this are said to *preserve determinism*.

The following corollary to our main theorem (Theorem 21) formulates concisely the backbone of our enhanced learning algorithm, which on demand refines the considered abstraction in an 'optimal' fashion, while in particular avoiding any problems due to non-determinism introduced by the abstraction (see Section 6.2). The corollary could also be proved as an independent theorem using lattice theoretic arguments. On the other hand, it is a direct consequence of the more constructive argument underlying the correctness proof for our enhanced partition refinement algorithm.

Corollary 1 (DCA: Alphabet Abstraction Refinement)**.** *Let $Sys = \langle Q, q, \Sigma_C, \delta, F\rangle$ be a DCA and $\alpha : \Sigma_C \rightarrow \Sigma_A$ a (finite abstraction) function. Then there exists an (abstraction) function $\alpha_{c,o} : \Sigma_C \rightarrow \Sigma_{c,o}$ refining α in a deterministic fashion with*

- $\exists \alpha_o \,.\, \alpha = \alpha_o \circ\; \alpha_{c,o}$ *and*
- *for all abstractions $\alpha_d : \Sigma_C \rightarrow \Sigma_d$ imposing a deterministic behavior on Sys, we have: $\exists \alpha_r : \Sigma_d \rightarrow \Sigma_A \,.\, \alpha = \alpha_r \circ \alpha_d \implies \exists \alpha_{r'} : \Sigma_d \rightarrow \Sigma_{c,o}.\; \alpha_{c,o} = \alpha_{r'} \circ \alpha_d$.*

The following section generalizes this theorem. This theorem, which works for countable Mealy machines (see below), is then proved in parallel with the presentation of the partition refinement algorithm for constructing the greatest determinism preserving abstraction in Section 6.2.

In order to show that such 'optimal' abstractions also exist in the slightly more complicated setting of Mealy automata, let us first pinpoint the essence of the difference. Rather than accepting words, Mealy automata produce an output after processing an (input) word. As in the case of DCA, we generalized the notion to allow countable sets of alphabets and states.

Definition 17. *A countable Mealy machine (CMM) is defined as a tuple $Sys = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda\rangle$ where*

- Q *is a countable nonempty set of* states*,*
- $q_0 \in Q$ *is the* initial state*,*

[1] Indeed, in practice one tests the real system to check for membership, and in most scenarios, the concrete system is supposed to have deterministic behavior.

- Σ *is a countable* input alphabet*,*
- Ω *is a finite* output alphabet*,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*, and*
- $\lambda : Q \times \Sigma \to \Omega$ *is the* output function*.*

Intuitively, a countable Mealy machine evolves through states $q \in Q$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(q, a)$ and produces an output according to $\lambda(q, a)$.

One can regard DCAs as CMMs whose output alphabet has just two symbols, one for acceptance and one for rejection. With this intuition in mind it is not surprising that the corresponding corollary looks almost identical.

Corollary 2 (CMM: Alphabet Abstraction Refinement)**.** *Let $Sys = \langle Q, q_0, \Sigma_C, \Omega, \delta, \gamma \rangle$ be an deterministic CMM and $\alpha : \Sigma_C \to \Sigma_A$ an arbitrary (abstraction) function. Then there exists an (abstraction) function $\alpha_{c,o} : \Sigma_C \to \Sigma_{C,o}$ refining α in an deterministic fashion with*

- $\exists \alpha_o \,.\, \alpha = \alpha_o \circ \alpha_{c,o}$ *and*
- *for all abstractions $\alpha_d : \Sigma_C \to \Sigma_d$ imposing a deterministic behavior on Sys, we have: $\exists \alpha_r : \Sigma_d \to \Sigma_A \,.\, \alpha = \alpha_r \circ \alpha_d \implies \exists \alpha_{r'} : \Sigma_d \to \Sigma_{c,o}.\ \alpha_{c,o} = \alpha_{r'} \circ \alpha_d$*

We will provide a partition refinement algorithm for the construction of the desired deterministic abstraction function via abstraction refinement.

6.2 Automated Alphabet Abstraction Refinement

In this section, we develop our partition refinement-based algorithm for alphabet abstraction refinement in the setting of Mealy machines, the system model we consider most adequate for practical applications. In this setting, we fix the output alphabet Ω (in the DCA case just 'accept' and 'reject'), and, given some initial abstraction Σ_A of the input alphabet Σ_C, we learn the coarsest abstraction that refines Σ_A and preserves determinism.

Our learning algorithm works directly on a representation system R for α-equivalence, i.e., initially, for each symbol of Σ_A we have a unique symbol of Σ_C, its concretization, which is used to query the concrete system Sys, whenever its abstraction appears in a membership query. This way, membership queries will never encounter non-determinism. This problem only arises when handling counterexamples. They can contain arbitrary alphabet symbols of Σ_C. Non-determinism can then be detected, when the counterexample transformed to an α-equivalent word consisting only of symbols in R produces a different output, as shown in Fig. 6.2. Here, the topmost line depicts the concrete input sequence defining the counterexample and the second line its transformation into the corresponding α-equivalent input sequence in R^*.

This is the point where classical learning would simply fail, and where our partition refinement-based technique shows its power. Key to our technique is the *'semantic'* way of maintaining a representation system during the refinement-based evolution of the input alphabet in terms of *witnesses*:

Definition 18 (Witness)**.** *Let Sys be a Mealy machine, $p, d \in \Sigma_C^*$, $c, c' \in \Sigma_C$. We call $(p, c|c', d) \in \Sigma^* \times \Sigma^2 \times \Sigma^*$ a witness (of the inequivalence of c and c'), iff*

$$\lambda'(p \cdot c \cdot d) \neq \lambda'(p \cdot c' \cdot d),$$

where $\lambda'(w)$ denotes $\lambda(\delta'(q_0, w))$ defined by $\delta'(q, aw) = \delta'(\delta(q, a), w))$.

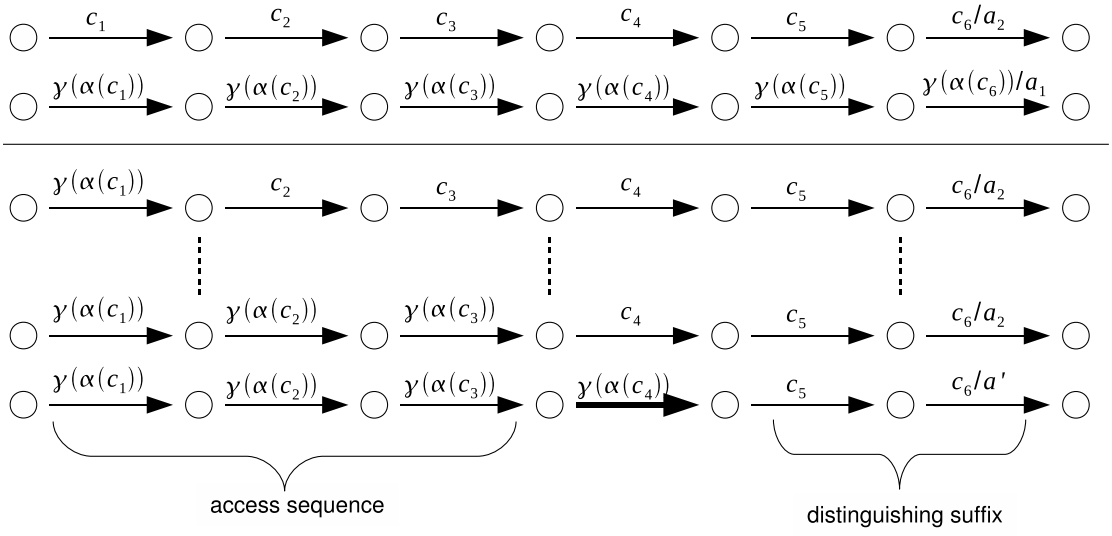

Figure 6.2: Processing of counterexamples

Witnesses form a *middle-congruence*. This middle-congruence can be used to refine the alphabet in the same fashion as the (Nerode) right-congruence is used to refine the set of states. We have:

Lemma 19. *Every counterexample exposing non-determinism can be decomposed into a prefix p, a concrete symbol $\bar{c}$ and a suffix d, such that*

$$(\gamma(\alpha(p)),\ \bar{c}|\gamma(\alpha(\bar{c})),\ d)$$

is a witness.

Sketch of Proof: Let us now assume a counterexample exposing non-determinism, i.e., of the kind as indicated by the first two lines of Fig. 6.2, where the symbols in each column are assumed to be α-equivalent. Then we proceed as indicated by the lower part of Fig. 6.2 by successively replacing the input symbols from left to right according to the following pattern:

$$\lambda'(\gamma(\alpha(p))\ \cdot\ \bar{c}\ \cdot\ d)\ \ =\ \ \lambda'(\gamma(\alpha(p))\ \cdot\ \gamma(\alpha(\bar{c}))\ \cdot\ d)$$

until the test fails, which is guaranteed to happen, because the fully transformed counterexample has a different output symbol. In Fig. 6.2, e.g., the replacement of the third counterexample input by its α-equivalent standard representative still works, but replacing the fourth input leads to an observable change of the output (a_2 becomes a'): There exists an index i, such that
$(\gamma(\alpha(c_1 \ldots c_{i-1})),\ c_i|\gamma(\alpha(c_i)),\ c_{i+1} \ldots c_m)$ is a witness. □

The witnesses found by counterexamples will become the key to the semantic refinement of α and γ:

$$\alpha_{new}(c) = \begin{cases} \alpha_{old}(c) & \text{if } c \notin \alpha_{old}(\bar{c}) \\ \alpha_{new}(\bar{c}) & \text{if } \lambda'(\gamma(\alpha(p))\ \cdot\ \bar{c}\ \cdot\ d) \\ & = \lambda'(\gamma(\alpha(p))\ \cdot\ c\ \cdot\ d) \\ \alpha_{new}(\gamma_{old}(\alpha_{old}(\bar{c})) & \text{if } c \in \alpha_{old}(\bar{c}) \setminus \alpha_{new}(\bar{c}) \end{cases}$$

$$\gamma_{new}(a) = \begin{cases} \gamma_{old}(a) & \text{if } a \neq \alpha_{old}(\bar{c}) \\ \bar{c} & \text{if } a = \alpha_{new}(\bar{c}) \\ \gamma_{old}(\alpha_{old}(a)) & \text{if } a = \alpha_{old}(\bar{c}) \wedge a \neq \alpha_{new}(\bar{c}) \end{cases}$$

As indicated by the term partition refinement, the semantic refinement of α only splits the equivalence class $\alpha_{old}(\bar{c})$, and abstracts all other concrete symbols as before (first line in the definition of $\alpha_{new}(c)$). The class $\alpha_{old}(\bar{c})$ itself splits into

- one subclass for all concrete symbols that behave like $\bar{c}$ on the witness (second line in the definition of $\alpha_{new}(c)$), and
- a second subclass for all the remaining elements of $\alpha_{old}(\bar{c})$ (third line in the definition of $\alpha_{new}(c)$).

The definition of γ_{new} arises then straightforwardly as shown in the definition of $\gamma_{new}(c)$. After each such refinement step, which adds an abstract symbol whose equivalence class is concretely represented by $\bar{c}$, the learning procedure continues by establishing closedness and consistency.

Thinking in terms of partitions and partition refinement is the key for proving the optimality of our alphabet abstraction algorithm. Let therefore

- $Part(\Sigma_C)$ and $Part(\Sigma_C^*)$ be the set of all partitions over Σ_C and Σ_C^*, respectively, and,
- for any $p, d \in \Sigma_C^*$, $c, c' \in \Sigma_C$ let $Ref_{(p,c|c',d)} : Part(\Sigma^*) \rightarrow Part(\Sigma^*)$ be the function that refines any partition over Σ_C^* according to a witness $(p, c|c', d)$ as described above. Furthermore, let
- $\alpha : \Sigma_C \rightarrow Part(\Sigma_A)$ be an arbitrary abstraction function, and
- $Part_d$ be the set of all partitions over Σ_C that refine the partition induced by $\equiv_\alpha$ and at the same time preserve determinism.

Then we have:

Lemma 20. *Let Sys be a system and $(p, c|c', d)$ be a witness. Then being an upper bound for $Part_d \subseteq Part(\Sigma_C)$ is invariant under the application of $Ref_{(p,c|c',d)}$.*

Proof: Let P be an upper bound for $Part_d$ and Q be an arbitrary element of $Part_d$. Then we can prove the required $Q \subseteq Ref_{(p,c|c',d)}(P)$ by contraposition as follows: $(z_1, z_2) \notin Ref_{(p,c|c',d)}(P)$ implies $(z_1, z_2) \notin Q$.
Let therefore $z_1, z_2 \in \Sigma_C$ with $(z_1, z_2) \notin Ref_{(p,c|c',d)}(P)$ and w.l.o.g. with $(z_1, z_2) \in P$. This means that z_1 and z_2 must have been split by $Ref_{(p,c|c',d)}$ and therefore that w.l.o.g. $\lambda'(p \cdot z_1 \cdot d) = \lambda'(p \cdot c' \cdot d)$ and $\lambda'(p \cdot z_2 \cdot d) \neq \lambda'(p \cdot c' \cdot d)$. Thus $\lambda'(p \cdot z_1 \cdot d) \neq \lambda'(p \cdot z_2 \cdot d)$. Thus, together with $Q \in Part_d$ we obtain as desired $(z_1, z_2) \notin Q$. □

This theorem obviously implies that every abstraction constructed during our alphabet abstraction refinement procedure induces an upper bound for $Part_d$.

If we now assume that we have a perfect equivalence oracle (which is standard for classical automata learning), we can combine this result with the correctness result for classical learning in order to obtain:

Theorem 21 (Optimality relative to perfect Equivalence Oracles)**.** *Let Sys be a deterministic system and α an abstraction on the input alphabet of Sys. Moreover, let us assume that we have a perfect equivalence oracle. Then we have upon termination of our refinement enhanced learning procedure:*

- *The last alphabet refinement ended at the coarsest refinement of α that preserves determinism.*
- *The learned automaton is behaviorally indistinguishable from Sys under the computed refined abstraction of the alphabet. In particular it can be used to reliably predict the Sys output for any (abstract) input word.*

This result shows that our elaboration of automata learning is very much in line with the also partition refinement-based approach of L^*. Like there, the correctness depends on a reliable equivalence oracle or counterexample finder. In addition, all the intermediately constructed hypotheses are optimal in the

sense that they concisely reflect all the considered observations: they are the state minimal automata (here Mealy machines) labeled with elements of a coarsest alphabet abstraction which are consistent will all the currently available observations. The formal proof of this statement follows a straightforward pattern for partition refinement algorithms. For many practical applications, like e.g., test generation, regression testing, or (behavioral) specification mining, this optimality result is both very valuable, and the best one could expect.

Similar to classical learning, where the complexity is always considered relative to the number of states of the final system Q, we also considered the complexity relative to the final size of the abstract alphabet Σ_A^f. As every round introduced by handling a counterexample either introduces a new state or a refinement of the alphabet, one can easily conclude that the algorithm terminates after at most $|Q| + |\Sigma_A^f|$ rounds, which is at the same time the maximum number of required equivalence queries.

Membership queries are required to complete the observation table and to treat counterexamples. The treatment of counterexamples will cause at most one membership query for each of its steps. Thus, the number of membership queries for treating counterexamples can be estimated by $m \cdot (|Q| + |\Sigma_A^f|))$, where m is the maximal length of a counterexample. The size of the final observation table can be estimated by $O(|\Sigma_A^f| \cdot |Q|^2)$, cf. [93, 98]. Accordingly, the total number of required membership queries is at most $|\Sigma_A^f| \cdot |Q|^2 + m \cdot (|Q| + |\Sigma_A^f|)$. In summary we obtain:

Theorem 22 (Complexity)**.** *Assuming that equivalence and membership queries both have some constant cost, our new learning algorithm has an overall complexity of:* $O(|\Sigma_A| \cdot |Q|^2 + m \cdot (|Q| + |\Sigma_A|))$.

Under the very reasonable assumption that the maximum length of the counterexamples m does not grow faster than the number of states $|Q|$, our complexity result reduces to $O(|\Sigma_A| \cdot |Q|^2)$, exactly the result known for classical automata learning. It should be noted, however, that this result, besides suppressing constant factors as usual in classical complexity theory, is also based on the assumption of constant time membership and equivalence oracles. Having in mind that equivalence oracles may not be realizable at all in practice, this might look quite unrealistic. Experiences made in the context of the ZULU challenge [42] though suggest that often equivalence oracles may be substitutable by fast counterexample finders. In fact, we were able to reduce the effort for realizing such a counterexample finder to very few membership queries [59]. This shows the potential of practical approaches and heuristics and it was one of the motivating observations that led us to founding the RERS initiative for experimental automata learning [36].

It should also be noted that the termination of our algorithm does not necessarily require Σ_C to be finite or Sys to be regular: It is sufficient that there exists a regular deterministic abstraction of Sys, as is illustrated in the next section.

6.3 An Example Run of the Algorithm

In this section, we will apply the method for abstraction refinement to a basic example to illustrate the integration with classical automata learning for Mealy machines (cf. [79, 98]). The pseudo code for our example is given in Fig. 6.3. It specifies a protocol entity that to the next upper layer provides primitives to receive and indicate messages and to the next lower layer exposes primitives to receive messages and to send acknowledgements. The component internally uses an continually increasing Integer variable to keep track of sequence numbers assigned to messages. Due to the counter, this system has infinitely many states even if we abstract from the content of the messages. We will show, how our algorithm refines the input alphabet just enough to reveal its Alternating Bit Protocol like stop-and-go behavior, which is regular.

Initially we assume an abstraction that has two concrete representative elements $msg(0, d)$ and $recv$, where msg and $recv$ denote two different primitives, and d some concrete instantiation of data. The abstraction refinement algorithm then is assumed to be able to partition the set of all possible inputs to the system according to this abstraction.

The learning algorithm will use an Observation Table and initialize the set of distinguishing suffixes D

```
enum {msg,recv} event;                          // events to occur
pdu p, ack;                                     // pdus
packet buffer = 0;                              // incoming data
seq_nr expect = 0;                              // next expected seq. nr
while (true) {
    wait_for_event(&event);                     // wait for event
    switch (event) {
    msg:
        from_lower_layer(&p);                   // read new message
        if (buffer==0 &&                        // if buf empty and
                (p.seq % 2 == expect % 2)) {    // and seq matches
            buffer = p.data;
            expect++;                           // increment exp. seq. nr
            indicate_to_upper_layer();          // indicate new data
        }
        break;
    recv:
        if (buffer==0) break;                   // skip if buffer emtpy
        data_to_upper_layer(&buffer);           // forward data
        ack.seq = (expect-1) % 2;               // ack. delivery
        to_lower_layer(&ack);
        break;
    }
}
```

Figure 6.3: Pseudocode of protocol entity

-	msg(0,d)	recv
ϵ	ind	-
msg(0,d)	-	ack(0)
msg(0,d) recv	-	-
recv	ind	-
msg(0,d) msg(0,d)	-	ack(0)
msg(0,d) recv msg(0,d)	-	-
msg(0,d) recv recv	-	-

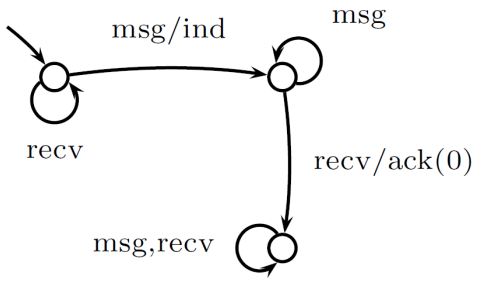

Figure 6.4: Observation Table and hypothesis at the end of the first learning phase

as $\{msg(0,d), recv\}$, the set of access sequences S as $\{\epsilon\}$ and the set of continuations SA accordingly. During the first round of learning the algorithm will find two additional access sequences $msg(0,d)$ and $msg(0,d)recv$. Fig. 6.4 shows the resulting observation table and the resulting abstract hypothesis.

Now, let the three-lettered word $msg(72,d')\ recv\ msg(73,d'')$ be the counterexample that is provided by the equivalence oracle. We first apply the current abstraction component-wisely to the counterexample. This results in the abstract word $msg\ recv\ msg$, which will be concretized to $msg(0,d)\ recv\ msg(0,d)$. The original counterexample and its representative will, when run on the system, lead to different outputs:

$$\begin{array}{lllllll} msg(72,d') & recv & msg(73,d'') & : & ind & ack(0) & ind \\ msg(0,d) & recv & msg(0,d) & : & ind & ack(0) & - \end{array}$$

At this point classic automata learning and a static abstraction would fail. The too coarse abstraction produces a conflict in observations that would be interpreted as result of inherent non-deterministic behavior or without interpretation simply as a failure of the experimental setup. Here, the proposed abstraction refinement method comes into play. The corresponding detailed analysis of the counterexample is shown in Fig. 6.5. The prefixes of the counterexample will component-wisely (one symbol at time) replaced by their

candidate			$reaction$			processing
msg(72,d')	recv	msg(73,d")	ind	ack(0)	ind	-
msg(0,d)	recv	msg(73,d")	ind	ack(0)	ind	replace *msg(72,d')* by $\gamma(\alpha($*msg(72,d')*$))$
msg(0,d)	recv	msg(73,d")	ind	ack(0)	ind	replace *recv* by $\gamma(\alpha($*recv*$))$
msg(0,d)	recv	msg(0,d)	ind	ack(0)	-	replace *msg(73,d")* by $\gamma(\alpha($*msg(73,d")*$))$

Figure 6.5: Treatment of a counterexample

according representative elements. The resulting word is then executed on the system. At one point the system's behavior will switch from producing the same output as for the original counterexample to producing different output. In this case replacing $msg(73, d'')$ will result in a different (conflicting) observation. We thus use the transformed prefix and the original element at the current position of the counterexample to refine the abstraction on the alphabet. The abstract symbol msg will be split into the abstract symbols $msg0$ and $msg1$. The witness revealing the difference is $(msg(0, d)\ recv,\ msg(0, d)|msg(73, d''),\ \epsilon)$.

After processing the counterexample is completed, the newly found representative element is passed to the learning algorithm as a new alphabet symbol. Assuming the learning algorithm will use this symbol only to extend the set of continuations SA and not in the set D as well (which is sufficient), the learning algorithm will after a second round terminate with a hypothesis that is equivalent to the behavior of the actual system. The resulting observation table and hypothesis are shown in Fig. 6.6.

6.4 Conclusion

We have presented an on-the-fly method for refining a given abstraction to automatically regain a deterministic behavior, a must for active learning. With this method detected non-determinism does no longer lead to failure, but to a dynamic abstraction refinement. Like automata learning itself, this method is in general neither sound nor complete, but it also enjoys similar convergence properties as long as the concrete (not necessarily finite) system itself behaves deterministically. From a practical perspective, our method allows users to 'experimentally' try abstractions and to let the algorithm care for the determinism requirement.

Our experience with a prototypical implementation of the enhanced learning algorithm is quite promising. We applied it, e.g., to the case study discussed in [9]. While Aarts et al. used a priori knowledge and hand tailored an abstraction in their case study in several iterations, we could simply provide a far too coarse initial abstraction, which was then automatically refined by our algorithm to terminate with the same result.

-	msg(0,d)	recv
ϵ	ind	-
msg(0,d)	-	ack(0)
msg(0,d) recv	-	-
msg(0,d) recv msg(73,d")	-	ack(1)
recv	ind	-
msg(73,d")	ind	-
msg(0,d) msg(0,d)	-	ack(0)
msg(0,d) msg(73,d")	-	ack(0)
msg(0,d) recv msg(0,d)	-	-
msg(0,d) recv recv	-	-
msg(0,d) recv msg(73,d") msg(0,d)	-	ack(1)
msg(0,d) recv msg(73,d") recv	ind	-
msg(0,d) recv msg(73,d") msg(73,d")	-	ack(1)

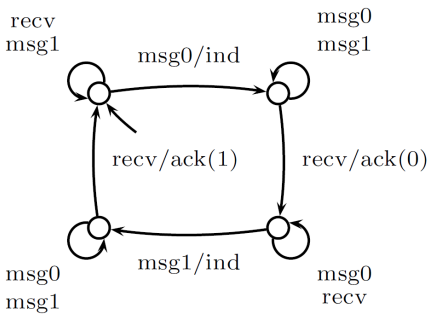

Figure 6.6: Observation Table and hypothesis after termination

7 Effect-Directed Learning

Within the context of the CONNECT project, automata learning is employed in ways that diverge considerably from, e.g., automata learning in the context of system validation. It thus makes sense to evaluate the requirements on automata learning techniques in a CONNECT-specific usage scenario.

Ultimately, automata learning in the context of CONNECT has to produce models suitable for connector synthesis, i.e., all system properties relevant for creating connectors have to be represented in the learned model. Most systems have preconditions regarding the observability of some system features. In, e.g., an e-commerce application, there usually is the precondition that a user has to be logged in before any business transactions can be conducted. Any model containing observations on features with such preconditions will also contain observations on their preconditions. This means that preconditions missing in the model imply that that system features are missing as well.

To ease the task of CONNECTor synthesis, it is desirable that the learned models only contain information on such system properties which are needed for proper operation of the CONNECTed system. This implies that a careful balance between (feature) completeness and model compactness has to be found.

The following sections outline these considerations in more detail and propose a concept of "effect-centric" learning, which aims at decreasing time spent on learning and keeping the learned model compact, while still guaranteeing a certain concept of "completeness", i.e., that the models contain relevant information. A short example illustrates the effectiveness of the approach, which has yet to be fully explored and implemented.

7.1 CONNECT Model Requirements

To better understand the requirements adherent to learning relevant (in the context of the CONNECT project) behavioral models for networked systems, let us first describe the communication, which we assume to happen between a pair of networked systems that are explicitly designed and developed to interact with each other properly. Figure 7.1 schematically shows the scenario: Two components communicate via protocol messages (1),(5). The components together realize some protocol. Both components are actual implementations of their specified interfaces. Without giving a formal definition, we can imagine both parties to comprise a control part (2), and a data part (3). The control part can be imagined as a labeled transition system with actual blocks of code labeling the transactions (4). Each code block in the components of Fig. 7.1 would consist of

- an entry point for one interface method,
- conditions over parameters and local variables,
- assignments and operations on local variables,
- a return statement.

The data parts may be best described as a set of local variables. We will refer to such a set of local variables as a *parameter structure*.

To infer the behavior of one component (e.g., the right one from Fig. 7.1), the part of the other component has to be taken by a learning algorithm, which will be equipped with the interface alphabet of the component to be learned. Obviously, in practice it can be prohibitively expensive to present a solution to the problem of generating a model that captures all the behavior defining the component. This task would correspond to reverse-engineering the component on the basis of its behavioral profile. Fortunately, the models required in the CONNECT project are of a different kind.

While, usually, models produced by active learning are used in model based verification or some other domain that requires complete models of the system under test (e.g., to prove absence of faults), in CONNECT, the inferred models will be used to explain how to interact with the system. This special focus allows us, to apply effect-oriented learning techniques (and the goal here is gathering enough information

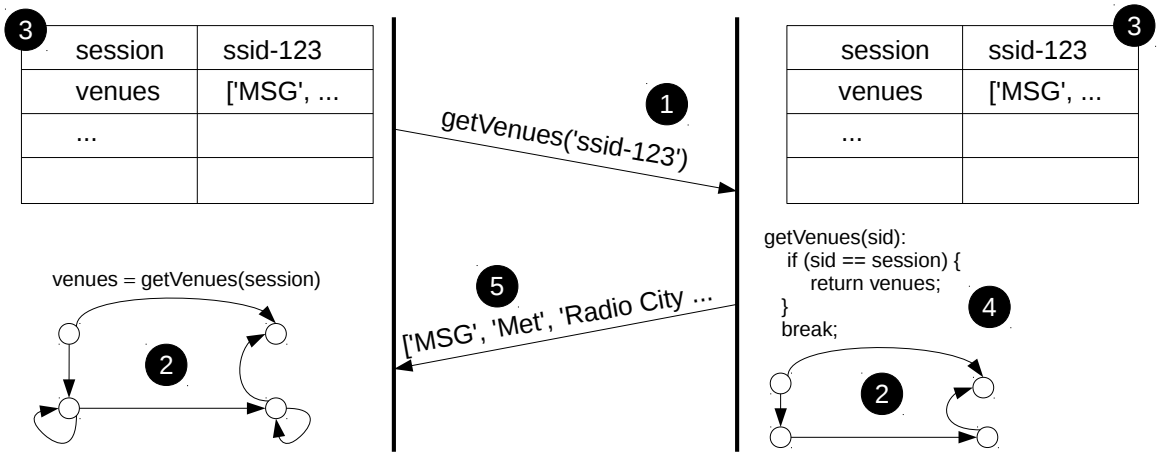

Figure 7.1: Communicating Components

to interact with a component successfully, i.e., to induce all desired effects on the target system): from the CONNECT perspective, complete models are rather uninteresting and would make the synthesis of CONNECTor models unnecessarily expensive. This may have a positive impact on the costs of producing models. However, to achieve this (capturing how to interact successfully), two kinds of information will have to be described by the models:

Effects of Primitives: The learned models will only be useful for CONNECTor synthesis within a given semantic context (cf. [64]). Most networked systems have well-defined purposes (or *effects*), e.g., electronic commerce systems. A subset of the offered communication primitives, when certain preconditions are met, will lead to successful conclusion of transactions directly relating to the respective purpose. There may, e.g., in a vending system, be a "purchase" communication primitive, that, when providing additional information like a product identifier, will conclude the process of buying a desired product. Other communication primitives may not directly lead to a successful conclusion of a business transaction, but may, e.g., be prerequisites for communication primitives serving the immediate purpose of the given system. A vending system may, e.g., have a communication primitive that results in the delivery of a list of products identifiers, which is a prerequisite for the "purchase" primitive.

It will be necessary to capture in the produced models, at which points in the model what effects are achieved (e.g., when a seat is actually booked in a system). These effects, however, will in general not be observable in the communication with a system; the information about effects of primitives rather has to be provided as an additional input to the learning algorithm, e.g., in an ontology.

Preconditions of Primitives: *(Data Causalities)* Many systems of interest for the CONNECT project operate on communication primitives which contain data values relevant to the communication context and having a direct impact on the exposed behavior. One example would be session identifiers or sequence numbers which are negotiated between the communication participants and included in every message. The models will have to make explicit causal relations between data parameters that are used in the communication (e.g, that always the exact session identifier that was returned when opening a new session has to be used in subsequent calls).

Sensible interface alphabets enable inference of data-related behavior.[1] While in classical automata learning these causal relations would implicitly be encoded in the state-space, there exist already methods to make such preconditions in the model explicit. [23, 24].

[1] Active learning classifies states and transitions by the output that is produced by a system. We assume here that the information about success or failure of the invocation of a primitive can be used as classifying output.

Without formalization, this will lead to models with transitions, like the one in Fig. 7.2, which

- are labeled with statements, the left component in Fig. 7.1 would execute,
- expose the causal influences of parameter values in form of preconditions, resembling the conditions in the component of the right side of Fig. 7.1,
- are annotated with the corresponding effects.

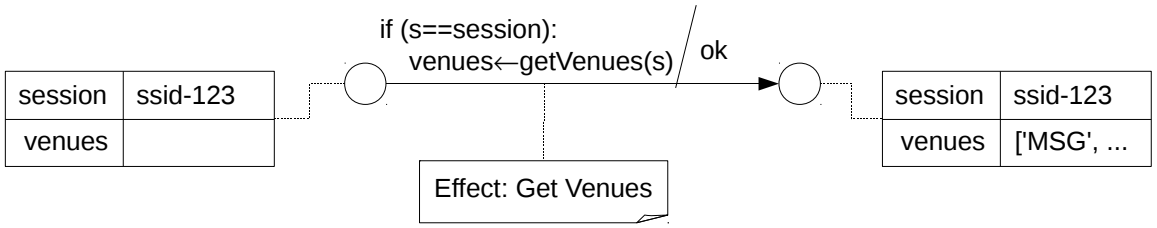

Figure 7.2: Transition in Inferred Model

Models comprising preconditions and effects for a given primitive will make it possible to generate sequences of communication primitives that (i) fulfill preconditions successively and (ii) finally conclude a process reaching a desired effect. This can be directly used for code synthesis. Also, reachability of the effects can be checked in the learned models: if a given effect cannot be reached this may be an indication that the learning process has not yet explored enough parts of the system. On the other hand, if all effects are reachable the learning process may be terminated safely, avoiding unnecessary further exploration.

7.2 Accelerated Learning and Effect Guarantees

Given a specification of effects to be achieved on the target system, it is possible to speed up the learning process considerably. The most simple approach, already hinted at in the previous section, is to terminate learning once all desired effects are reachable in the learner's current hypothesis. This induces only minimal changes in the learning setup and no changes to the core learning algorithms - only a component which evaluates the learned hypothesis with regards to the effects specification is needed.

A more integrated approach to effect-centric learning may employ learning algorithms exploring the target system in a fashion similar to directed search, e.g., an exploration heuristic may favor areas of the target system that already satisfy a number of preconditions. This is possible because the hierarchy of effects and preconditions induces a notion of progress in the sense that the exploration process is usually closer to successful termination the more preconditions have already been triggered and observed. It should be noted that learning techniques employing directed search methods only make sense in scenarios where complete exploration is not aspired. Thus any such approaches are legitimized by the, compared to, e.g., scenarios of system validation, narrower scope of the learned models to be produced in the context of CONNECT.

The introduction of effects induces fundamental changes in how equivalence approximations can work. Instead of trying to determine if a given model is a faithful image of the target system as a whole, the equivalence approximation can focus on validating paths in the learned model that supposedly achieve desired effects. Combined with effect-aware strategies during the exploration phase this means that automata learning is mostly concerned with finding paths that lead to desired effects (exploration) and then in succession safeguarding these paths (equivalence check). Additionally, model-checking techniques can be employed to, e.g., detect effects that were discovered without their preconditions. Any such occurrences are in apparent contradiction to the application-specific knowledge expressed by means of effects and thus should be reason for focused examination.

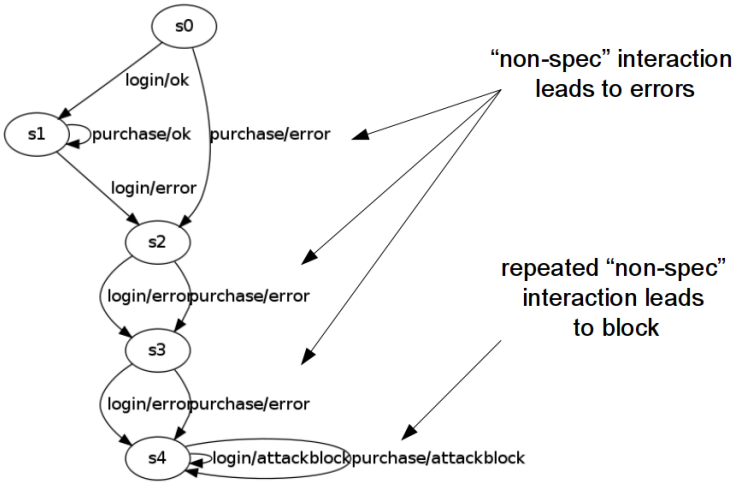

Figure 7.3: Behavioral model of a small example system

This shows how effect-centric learning may not just speed up the learning process by keeping a focus on relevant parts of the target system, but that the learning process can also take increased care to guarantee that all specified system-effects are indeed reachable by the learned model.

Aside from using knowledge on the hierarchy of preconditions and effects to efficiently create models useful for CONNECT scenarios, the learning process itself may be used to identify previously unspecified causality between effects. If, e.g., in a learned system model all paths to a certain effect include another effect, that effect may be identified as a possible precondition to the effect reached by the considered paths. Any such approach, however, must cope with the uncertainty stemming from the limited availability of samples supporting any given hypothesis on additional, previously unspecified preconditions: the more samples are available, the higher the confidence level attained.

7.3 Example

Let us consider a small example in which a minimalistic e-commerce application is the target system for the learning process. This system allows purchasing items after logging in. Any interaction with the target system lying outside the bounds of the application interaction specification (i.e., first login, then purchase) will be answered with an “error” message. Once in an error state, the system will refuse supplying its services to the client (which, apparently, has an invalid internal state causing the misbehavior), meaning a new communication session has to be initialized before normal operation can resume. If, however, after an error message the client continues with the ill-formed communication the target system will assume it is subject of a Denial of Service (DoS) attack and block further communication, including opening new communication sessions, with this client. A model of this system is displayed in Fig. 7.3.

The obvious main purpose of this system is to support the conclusion of e-commerce transactions, i.e., a client expects to observe a message that confirms that the “purchase” communication primitive was processed successfully. Thus, when denoting effects as pairs of input- and output-messages, the effect “purchase/ok” is a desirable one, as it indicates the successful conclusion of a business transaction. The effects “login/error” and “purchase/error”, however are clearly much less desirable. The “login/attackblock” and “purchase/attackblock” effects are downright undesirable and should even be avoided.

Given the desired effect “purchase/ok”, an effects-aware learning setup can, e.g., produce a model similar to the one presented in Fig. 7.4, which was learned with a preliminary implementation of effect-centric learning. It is obvious that this model does not cover the complete system behavior. It does, however, include all specified effects and thus can be considered “feature complete” in a sense relevant to the CONNECT project.

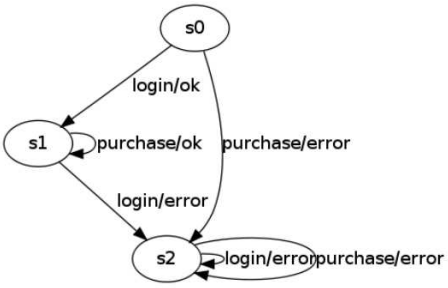

Figure 7.4: Learned model of the example system

In this example, the effects-aware learning procedure simply terminated learning when all specified effects were reachable in the learned hypothesis. Aside from only spending 14 membership queries (instead of 44 when learning the complete system), this also incidentally avoided tapping into a state showing the “attackblock” behavior, demonstrating that effects-aware learning may succeed in extracting a meaningful model of relevant system parts where full exploration of the system may fail instead.

The learned model may also be used to extract candidates to complete the hierarchy of effects and preconditions. In this example one may consider “login/ok” as a precondition for “purchase/ok”, as all paths leading to the latter include the former. However, given there is only one path leading to “purchase/ok” this conclusion calls for a considerable leap of faith.

7.4 Conclusion and Future Work

With effect-directed learning, the process of generating behavioral models can be accelerated considerably. Additionally, beyond mere performance improvements, effect-directed learning can help to guarantee a certain level of completeness wrt. expected features a system to be learned shall exhibit. This can help to ensure that learned model indeed are meaningful in CONNECT context, e.g., that synthesis has enough information to work successfully.

In Y2, effect-directed learning was identified as potentially promising, a finding backed up by initial investigations on overall usefulness by means of an early implementation. Moving forward, effect-directed learning has to be fully integrated into the learning toolchain and embedded into the CONNECT architecture. A sound formal structure has to be developed and applied.

8 Learning Non-functional Properties

In the CONNECT framework, the learning enabler is envisioned to gather information about non-functional properties while inferring functional models of networked systems. In this chapter we will introduce the necessary extensions to learning technology alongside some results of a prototypical application of this new method. Considering the CONNECT architecture and data-flow (cf. Fig. 1.1), the learning enabler will receive interface descriptions of networked systems, an ontology on the data domain, and metrics of interest provided by networked system from the discovery enabler. While the first two will be used by the learning enabler to construct a test-driver and abstraction(s) as discussed in the previous chapters, the metrics of interest will be used to generate fitting probes into the learning process (see below).

The learning enabler will produce *guarantees provided by the networked system* as (non-functional) output, which will be used during dependability analysis. Consider latency, i.e., response time as an example of such a guarantee: the learning enabler can provide the information that a networked system did react to some message within 20 ms in 95% of the cases and did always react within 200 ms. Further examples are throughput, failure rates or influence of load on the aforementioned metrics.

As part of WP4, the intention is to integrate learning and monitoring. While at first it may appear that gathering data on non-functional properties may be an obvious point of integration, the actual CONNECT data-flow shows that monitoring components are to be generated as part of the CONNECTor (cf. Section 9.2), which, obviously, is not available during behavioral learning. Monitoring is mostly concerned with collecting data on the CONNECTed systems, while collecting data on non-functional properties during learning will target the individual, still unCONNECTed systems. We will thus generate probes into the test-drivers generated in the course of (functional) learning.

8.1 Architecture of Extended Test-drivers

Let us briefly recapitulate the experimental setup of a learning process, which is shown in Fig. 2.2 of Chapter 2. The learning algorithm (resp. equivalence approximation algorithm) will formulate queries on an abstract level. The queries will be passed to a test-driver that translates queries into sequences of single actions on the system under test (SUT), e.g., message that are sent to the system. The test-driver will execute these actions one by one and record the respective results.

We can easily extend the test driver to collect some data during the active interrogation of the target system, which is part of the employed active learning methods. The data collected this way, however, might be strongly biased for three reasons:

1. During the learning process tests are not equally distributed over all states and actions of a system.
2. Learning may not produce enough tests to formulate solid statistical guarantees.
3. Learning usually is thought (and implemented) to work sequentially: first some tests are done, then a hypothesis is formulated and then some more tests are conducted.

This gives reason to believe that data obtained during the normal learning process may have limited quality when compared to methods that focus on providing non-functional data in the first place. The upside, however, is that this data can be collected without conducting interactions with the target system beyond what is needed for the behavioral exploration.

Issues (1) and (2) can be addressed by identical means: to produce sufficient number of well-distributed tests, a two-step approach can be employed, where in the first step a model of a system is learned. This is followed by extensive tests on the basis of this model. This setup coincides with the classical learning setup: first a model is learned, then a conformance test is run to approximate an equivalence query. We can thus tightly integrate tests run for non-functional data and the search for counterexamples. This can be based on the *evolving hypothesis* approach discussed in Chapter 3. Using this approach, it is

Algorithm 1 Learn Model

Require: $S = \{\epsilon\}$, $SA = \Sigma$, $D \subseteq \Sigma$

```
1: loop
2:   repeat
3:     if (∃sa ∈ SA)(∀s ∈ S) : row(sa) ≠ row(s) then
4:       S ← S ∪ {sa}
5:       SA ← (SA ∩ {sa}) ∪ ({sa} × Σ)
6:     end if
7:     complete row-vectors
8:   until closedness is established
9:   construct H
10:  if (∃c ∈ Σ*) : λ'_S(c) ≠ λ'_H(c) then
11:    exploit c
12:  else
13:    return H
14:  end if
15: end loop
```

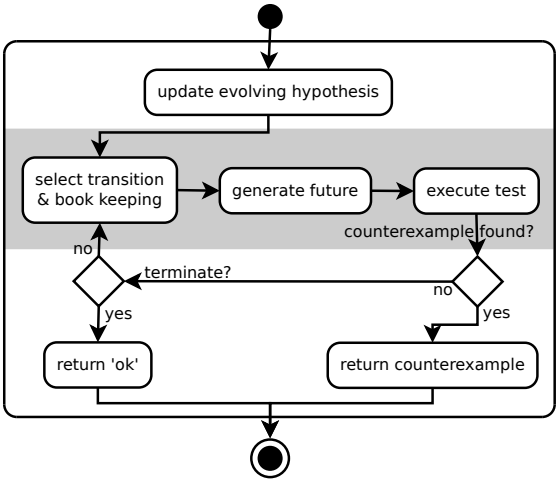

Figure 8.1: Parallelized learning / equivalence approximation

only necessary to extend the used transition/future selection and termination criteria to ensure statistical robustness.

The potential benefits of obtaining data with higher precision have to be weighted against the expenses for obtaining them, though, as producing large numbers tests conflicts with the goal of making learning available on mobile devices and establishing CONNECTions at need. The learning enabler will thus also collect non-functional data in the course of learning to provide at least rudimentary non-functional data in contexts with limited resources.

Issue (3) has to be addressed by different means. To produce a complete picture of the non-functional properties of a system, exposure to realistic load is needed. The only way to do this is conducting tests in parallel. It is easy to see how the setup used for equivalence approximation (cf. Chapter 3) can be parallelized: one can simply select multiple transitions at a time and conduct multiple tests per transition in parallel. However, as discussed, the extensive amount of tests conducted as part of an equivalence query may not be feasible in every scenario.

To produce realistic load in the learning phase, the learning algorithms in LearnLib [72] can be modified to no longer produce single queries but rather batches of queries. This can be achieved by handling all occurrences of, e.g., *unclosedness* at the same time (cf. Figure 8.1). Here, we will not discuss the technical details but rather present results of a prototypical application of the ideas. Exact calculations will be in the scope of next year's research. One drawback of the batches constructed during learning, however, is that the size of these batches usually varies in the process of learning. To conduct tests in a controlled environment (i.e., include the load in the test setup), one will have to use tests of the kind discussed above.

The discussed extensions to the learning process result in a setup that is shown in detail in Fig. 8.2. The learning algorithm (resp. equivalence approximation) produces batches of queries and passes these to an oracle, which distributes the queries to a number of test-drivers. The test-drivers consist of (1) a TestOracle, serving as interface to the learning algorithm, (2) a TestDriver, doing the translation from symbols to actions on the system, and (3) a SUTInstrumentation actually performing the actions on the SUT. We will add probes at different points. The TestDriver will record data about the execution of single actions (e.g., response times), while the TestOracle will record data on the level of complete queries as well as on the level of symbols. The TestOracle can, e.g., record failure rates for symbols. This is not possible in the TestDriver as it requires an interpretation of the obtained (or omitted) answers. This architecture, i.e., the combination of probes in the test process and parallel execution of tests in a two-step approach, allows us to provide data for a broad range of non-functional properties relevant in the scope of CONNECT.

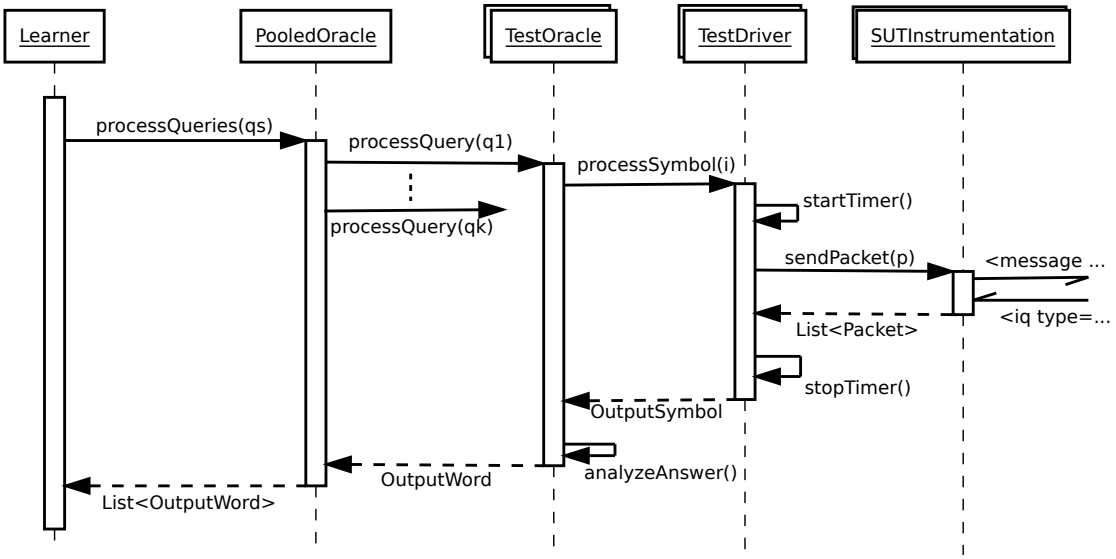

Figure 8.2: Sequence chart probe integration

8.2 The XMPP Case Study

We have applied the introduced learning framework for non-functional properties in a small case study, using prototypical implementations of the concepts outlined above. The XMPP protocol [104], which can be used for instant messaging, was used as an example as part of the ongoing integration work between packages. We used the extended test-drivers in the experiment that focused on the subscription part of XMPP. In XMPP users can subscribe to notifications about the status (presence) of other users.

The conducted experiment used two connections and two users that would interact through an XMPP server, each one using a dedicated connection. Both users were allowed to send *presence* notifications. User A was allowed to *subscribe* and *unsubscribe* to User B's presence information. User B had to confirm both, that A is *subscribed* and that A is *unsubscribed*. This resulted in a total of 6 messages as the alphabet for the learning algorithm (Authentication was omitted in this example).

The final model was learned using 186 tests in the learning phase and 679 queries in total. It took 704 seconds to complete. To produce realistic data, the XMPP server (openfire) was deployed on a server in Frankfurt (Germany), while the learning was done on a computer in the network of the TU Dortmund (distance of 200km). Communication was realized over the internet.

The resulting functional model for User A is shown in Figure 8.3. White packages are the ones User A can send and gray packages are the ones User A can receive. Packages sent and received by User B are hidden in this model. As expected, the system can be in states of approved and pending (un-)subscription.

The learning process made use of batches. Fig. 8.4 shows the size of batches during the learning and during the equivalence phase. In the equivalence phase the batch-size was fixed to 50 queries. A total of 500 queries was executed in 10 batches in the testing phase (some of these queries could be answered by a cache). This could be varied to produce data in relation to load. As expected, the batches produced in the course of learning vary in size and do not reach the size possible in the explicit testing phase.

The interaction with the XMPP server was realized by sending a message and then timed wait operations (of 1 second each) on each connection. The waiting would be repeated as long as answers were received. This resulted in an execution time of about 2 seconds per symbol. Usually, only a very small fraction of this time was needed to get a response. Table 8.1 shows response times for the actions of User A for the learning- and the equivalence approximation phase. A *subscribe* was in the average answered in 21.57 milliseconds during the learning phase and within 18.13 milliseconds during the equivalence (EQ) phase. For both messages the number of executed tests is much higher in the EQ phase. Only for the the *unsubscribe* message this lead to a smaller standard deviation.

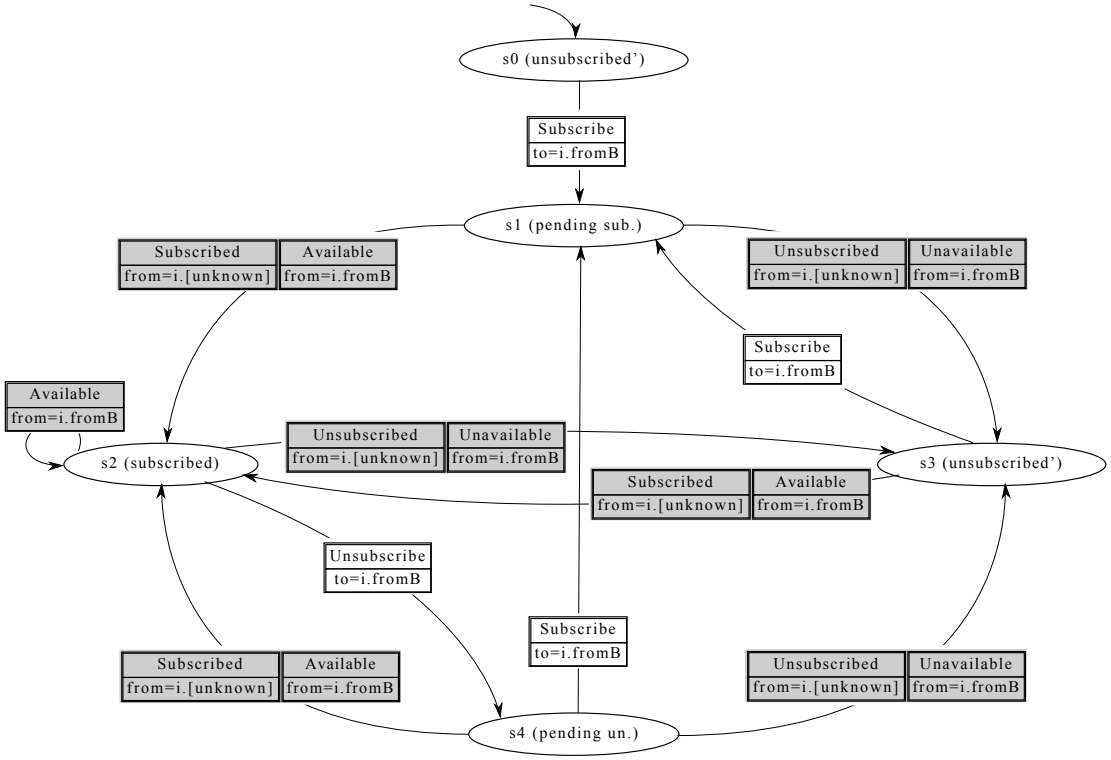

Figure 8.3: Behavioral model of subscription part of XMPP

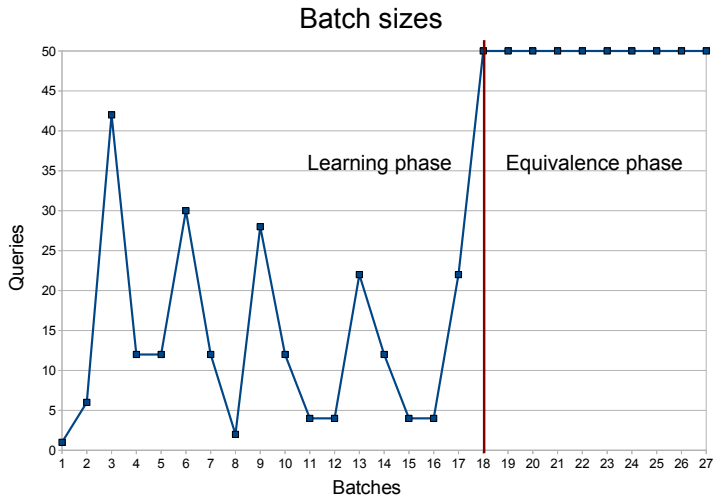

Figure 8.4: Batch sizes for XMPP case study

During these experiments there were no failing messages. In other experiments with bigger batches (more than 100 queries in parallel), however, we could observe failures. To produce solid guarantees for the dependability analysis, more tests would have been needed than have been conducted in this prototypical experiment. Such experiments would have to vary the size of batches and use a real coverage criterion and not an arbitrary limit of 500 tests. Basic non functional properties like an observed maximal response time, on the other hand, can be produced with this setup right away and used in other work packages.

Table 8.1: Response times for XMPP case study

Message	Phase	Average [ms]	Std. dev.	Max [ms]	# tests
subscribe	Learn	21.57	18.85	154	263
	EQ	18.13	24.83	229	1039
unsubscribe	Learn	8.56	26.42	272	125
	EQ	5.07	16.42	270	748

8.3 Conclusion and Future Work

The learning stage is in direct interaction with system under test during construction of the behavioral model of the target system. By augmenting the learning setup with means to gather statistical data, data on non-functional properties can be collected “for free”, without system interaction beyond what is needed for the behavioral exploration. Data obtained this way, however, may show statistical bias stemming from the fact that the queries during learning are focused on exploration. In scenarios allowing further system interaction, additional queries guided by the behavioral model can be created to even out any such bias.

A main focus for the coming months will be integration with other work packages. As mentioned above criteria have to be found to conduct a fitting number of tests to achieve a certain level of confidence. This directly relates to the desired integration with other packages, as consumers of data on non-functional properties should specify the confidence level wanted and the set of properties to be observed.

The parallelization of learning components is obviously useful beyond load tests, e.g., to speed up learning by querying several target systems in parallel. Thus parallelization should have deep-rooted support in the learning framework.

9 The CONNECT Runtime Monitoring Infrastructure

As we have seen in the preceding chapter, in CONNECT we need to continuously monitor the runtime behaviour of the CONNECTed systems to respond to the growing needs of evolution and adaptation. The events to be observed, belonging to guaranteed functional and non-functional properties, can themselves vary in scope and along time. To address such needs, in D5.1 we had already discussed the requirements for such a monitoring infrastructure in CONNECT. Such requirements are briefly summarised in the next section. In particular, we expand here on the requirements of modularity and flexibility, which are key to support the integration with the various CONNECT enablers.

In this chapter, we describe the implemented infrastructure and show how it can be applied to parameterized models for various types of runtime analyses.

9.1 The Place of Behaviour Monitoring in CONNECT

Coping with context changes requires that software be able to self-organize its structure and self-adapt its behavior to provide the desired quality of service. Assumptions taken by developers at design-time, in fact, may become invalid after the system is deployed and running and cannot fully take into account the changing external world interacting with the system. A runtime evaluation process is hence needed to timely detect unpredictable changes that affect the software system behavior.

This on-line analysis can involve non-functional aspects that need to be taken into account for addressing performance measurement and run-time performance management. In particular, runtime analysis is the basic step for collecting the execution-time values of specified parameters in order to fill program executions models or perform on-line performance prediction.

The runtime assessment process, as opposed to activities that are carried out in the development/-coding phase, can be identified as a *monitoring* activity. The monitoring process involves several different activities carried out during the execution of a system, dealing with data collection, interpretation and presentation of information concerning the “observable object”. Each of these activities is the topic of a heterogeneous literature and addresses specific problems and techniques providing different solutions (see a brief overview in Section 9.5).

The most commonly used approach for observing the behavior of distributed systems is event-based monitoring. In a large system a lot of events can occur, which need to be filtered and combined to detect unexpected behaviors and to estimate performance or dependability measures of the system. In this chapter we present a generic event-based monitoring infrastructure, called GLIMPSE[1], that implements the key components of a generic, flexible and robust architecture for composite event detection by means of publish/subscribe messaging pattern. Current event-based monitoring systems mainly focus on simple and composite event specification languages, but provide a limited flexibility in dealing with the large and dynamic context of monitor applications and the existing technologies. Our contribution is a highly flexible and lightweight architecture decoupling high-level event specification from the underlying observation and analysis mechanisms, thus yielding the greatest generality and facility of use.

The very vision of CONNECT, i.e., achieving automated and eternal interoperability, puts on-line approaches, and therefore monitoring, in a central position. Monitoring supports the construction of feedback loops whereby approaches to dependability analysis, CONNECTor synthesis, and behaviour learning can be applied to an on-line setting and can be enhanced to cope with change and dynamism.

Monitoring is performed alongside the functionalities of the CONNECTed System and is used to detect conditions that are deemed relevant by the other CONNECT Enablers. Upon detecting such conditions, the monitoring system alerts the interested Enabler which, in turn, triggers an update of the analysis, synthesis, learning respectively. In this way, powerful but expensive techniques are executed only when

[1] Generic fLexIble Monitoring based on a Publish-Subscribe infrastructurE
http://labse.isti.cnr.it/tools/glimpse

necessary.

Although monitoring can can provide valuable support to achieve these objectives, it can easily incur feasibility problems caused by excessive overhead. Also, as we intend to realise a monitoring system that can address different purposes (spanning functional and non-functional aspects), it must be designed with special emphasis on flexibility.

An evaluation of the requirements imposed on the monitoring subsystem (see D5.1) by the other elements of CONNECT has led us to identify several key features that a generic, flexible monitoring infrastructure should possess. Some of these are:

Distribution, dynamicity, heterogeneity: The monitoring system must be able to observe systems that are inherently distributed, highly dynamic and composed of heterogeneous entities that were not necessarily conceived for interoperation. Monitoring should address distribution by adopting an architecture that is itself distributed.

Modularity and Flexibility: A key goal of our work is to realize a monitoring infrastructure that can be employed for different purposes (including monitoring of functional an non-functional properties) and in different settings (even beyond its application within CONNECT). This is achieved by building our infrastructure around a modular architecture whose coupling with the observed system is minimal.

Efficiency: The performance penalty incurred because of monitoring should be minimized (and possibly predictable), while achieving the intended observation goals. Approaches for minimizing the impact of monitoring involve the mechanisms of monitoring (e.g., sampling, self-tuning [25] as well as its architecture (e.g., by improving scalability through a hierarchical organization [83]).

Predictable overhead: The approach to overhead reduction followed in most existing monitoring systems follows a best-effort policy, whereby overhead is kept as low as possible but is in fact unbounded, as noted by Callanan et al. [35]. We agree that the load caused by monitoring must not just be reduced, but also by predictable and controllable.

Model-driven approach: Model-driven and generative approaches [96] allow software engineers to express the key concepts of a domain or system at a high level of abstraction and then use them for reasoning and for automating coding tasks that are otherwise costly and error-prone. Most of the research effort carried out in CONNECT relies on some sort of models; it is therefore natural to pursue integration at an abstract conceptual level in order to benefit from automated techniques for deriving the actual implementation code.

Integrated with the other CONNECT Enablers: Although striving for genericity and flexibility, our framework will have the primary characteristic of being tailored after the specific needs of CONNECT and, in particular, will be matched with the other core functionalities provided by CONNECT Enablers. More on integration is discussed, although still at an abstract level, in the next three subsections.

9.2 Architecture

Monitoring has been defined as *the process of dynamic collection, interpretation, and presentation of information concerning objects or software processes under scrutiny* [66].

Furthermore, monitoring is applied in very different application scenarios and for different purposes, spanning system and network management, performance analysis, dependability assessment, security.

In [77], the authors identify the fundamental problems and issues related to the monitoring in distributed systems.

Elaborating on [77], five core functions can be identified for a generic monitoring system:

1. Data collection: this function concentrates on the collection of raw data from the execution of the observed components. This can be done by either instrumenting the subject component (when this is possible) or by intercepting interactions among components through a proxy-based probe. A special case is represented by the built-in logging facilities that many systems provide natively.

2. Local interpretation: this function refers to the filtering that raw data go through before being interpreted at an aggregated level, to remove redundant or irrelevant information, may be
3. Data transmission: in distributed systems, this function takes care of gathering information from different originating nodes to a central (possibly not unique) node. Data transmission might exploit smart optimization algorithms (e.g., to delay data transmission or to give higher priority to certain information, when the network is subject to congestion)
4. Global interpretation: (aka "correlation" or "aggregation"), this function makes sense of pieces of information that come from several nodes and puts them together in order to identify interesting conditions/events that can be observed only at an aggregated level. This function can be realized by means of a complex-event processing engine.
5. Reporting: this function deals with presenting the results of monitoring in a format that is meaningful to the "consumer" of the monitoring system. The consumer can be a human (e.g. a system administrator) or a program, e.g., a software component that implements the feed-back loop in a self-controlled software system.

GLIMPSE, the monitoring infrastructure used in CONNECT, was designed in order to cover these five functions in a modular, flexible way. GLIMPSE will be used to support behavioural learning, performance and reliability assessment, security, and trust management, however the infrastructure is totally generic and can be easily applied to different contexts.

During CONNECTed system usage, a huge amount of primitive events may be generated, the monitoring system must have a powerful filtering mechanism to process complex events and should minimize intrusiveness on the monitored system and processes.

GLIMPSE uses the classical observer pattern and has been designed according to the mentioned requirements of modularity and flexibility, so it can be improved and enriched, even at runtime, with components that respect a minimal set of requisites.

The usage of a messaging system in ESB allows GLIMPSE to use a variety of protocols such HTTP/-SOAP and REST; besides, with the usage of JBI [4] components, GLIMPSE may interact even with legacy systems, binary transports, document-oriented transports, and Remote Procedure Call systems, this solution reduces system bottlenecks that might occur on Remote Procedure Call [5] or Database-centric architecture.

The architecture of GLIMPSE, shown in figure 9.1, is composed of five main components:

Probes (Consumer/Enabler) Probes are realized by injecting code into a CONNECTor. We assume that every CONNECTor when synthetized, will be equipped with probes that summarizes and sends to the GLIMPSE Monitoring Bus primitive events when they occurs. The probes, may also be instructed to filter events in order to reduce the amount of raw data sent on the GLIMPSE Monitoring Bus.

Monitoring bus The monitoring bus is the communication backbone where all the informations (events, questions, answers) are sent on by: Probes, Enablers, Complex Event Processor and by all the services joining GLIMPSE.

The bus is created using a publish-subscribe paradigm and is driven by the Manager component; this pattern has been chosen to allow more users to fetch the same CEP evaluation results and to offer connectivity at low computational-cost.

Complex Event Processor Complex Event Processor (CEP), the core of the monitoring architecture is the rule engine where the primitive events, coming from the probes, are analyzed. There are several rule engine that can be used for this task, to achieve better decoupling from a specific rule language (like Drools Fusion [2], RuleML [6]), we are developing a generic rule language making a e-core metamodel to define rules. This e-core metamodel is presented in D2.X.

Temporarily, we used an XML language to define the action that the Enablers request to the Monitoring; the schema is described in the section 9.3.

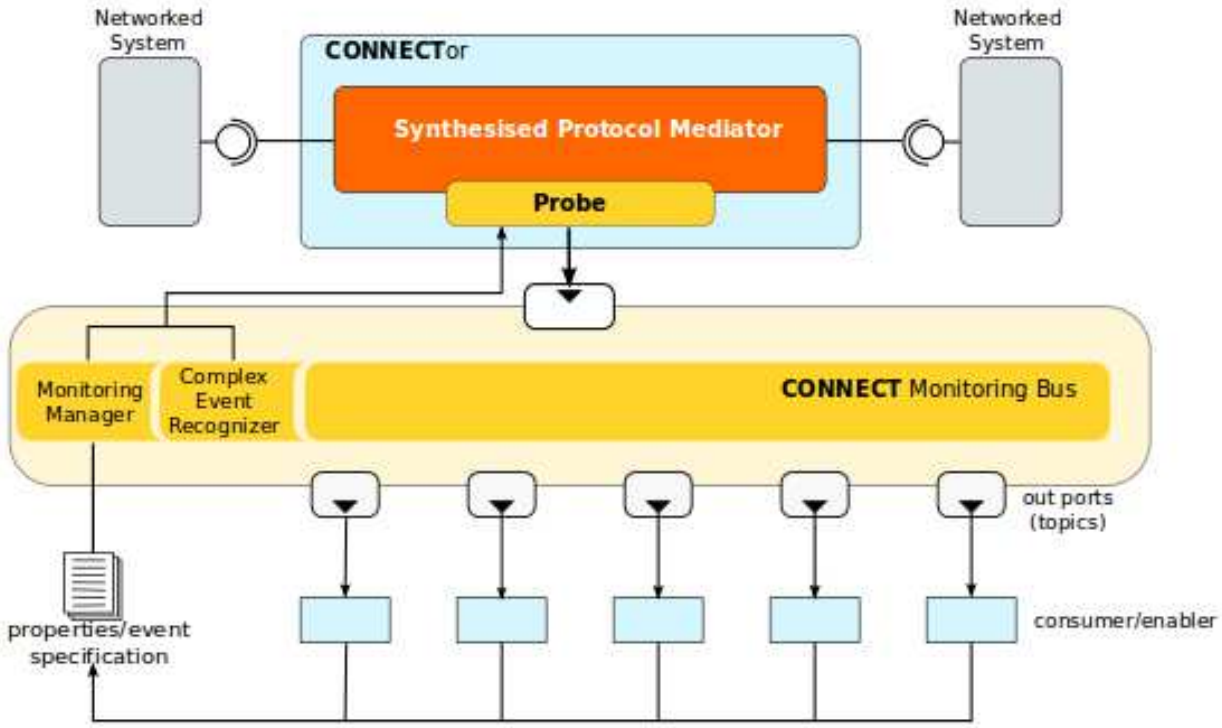

Figure 9.1: The architecture of GLIMPSE

Enabler It may be a learning engine, a dependability analyzer or a simple customer that needs to known non-functional information about CONNECTed system. It sends an evaluation requests for a property to the Manager using the Monitoring bus and waits for the evaluation results on the answer channel provided by the Manager. The request is expressed using the XML language specified in section 9.3.

Manager The Manager component is the orchestrator of all the GLIMPSE architecture. It start the rule engine (CEP) and manages all the communications. Listen for the Enablers requests, creates and provides the response channel.

9.3 Implementation

We have developed a prototype of GLIMPSE following the architecture described in Figure 9.1. GLIMPSE implements the five core functions identified in Section 9.2 in this way:

Data collection is entrusted to the Probes that collect data and send it on the Monitoring Bus using a JMSMessage. The Event occurrence is identified with the label of the LTS transition fired; this value is encapsulated into the data field of a ConnectBaseEvent object. The JMSMessage contains also some extra properties, such as SENDER, to define who is sending the request and RECEIVER, to define who must receive this message. The payload of the JMS Messages sent on the bus is an implementation of ConnectBaseEvent interface, defined in figure 9.3.

In the CONNECT framework we suppose a non-malicious behaviour of the CONNECT enablers; for this reason, these extra properties can be used without any sort of security or trust evaluation. The timestamp of each event sent by the probes is set when the event is created. We monitor the behaviour of the CONNECTor; for this reason, it is not necessary to have mobile code or to deploy part of the CONNECTor on the networked system.

Local interpretation is also carried out by the Probes. Probes may be able to receive messages

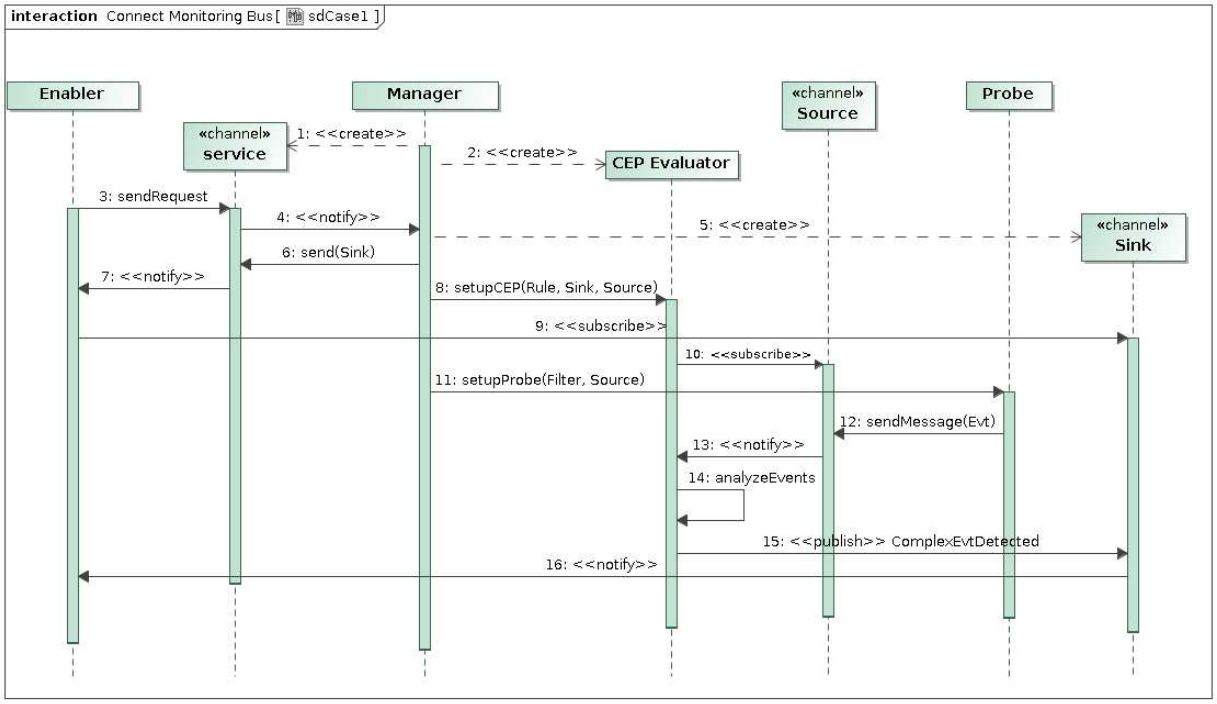

Figure 9.2: Sequence diagram of a classical interaction into GLIMPSE Monitoring Bus

containing the filter to apply on the sending event task, this proactive action may decrease the number of messages (events) to send on the bus. These filter parameters may be sent by the Manager component after receiving a request from an Enabler.

Data transmission task is realized by the system backbone (Monitoring Bus), implemented by ServiceMix4 [7], an OpenSource Enterprise Service Bus, used to combine advantages of Event Driven Architecture and Service Oriented Architecture functionality.

We choose ServiceMix4 because it offers a Message Oriented Bus and runs a JMS-compliant component like ActiveMQ [1] as messages broker. These components are interchangeable without any sort of impact on the rest of architecture.

Global interpretation is provided by the CEP Evaluator component, implemented using JBoss Drools Fusion, a rule engine for Complex Events based on Charles Forgy's Rete algorithm [45], chosen for the useful integration on ServiceMix4, and for the powerful and expandible rule language. The CEP is able to make analysis examinating the stream of events present on the Monitoring Bus following the rules provided by Enablers; it is configured to work with a stream of events and allows Enablers to set up techniques such as detection of complex pattern or relationship between timing, overlaps, hits of events within a temporal cloud. It fetches messages coming from Probes (Data collection) on the Monitoring Bus (Data transmission), analyzes them, and answers on the sink channel provided by the Manager; the CEP is able to reconstruct the temporal order according to which the events arrive on the bus, but if there are delays on the transmission from the probe to the source channel the sending order will not be respected, for this reason on the ConnectBaseEvent we included the field SequenceID.

Reporting action task is implemented by the Manager and by CEP Evaluator. The Manager fetches messages coming from Enablers on the service channel, analyzes them, instructs Probes if necessary, startups or updates the CEP Evaluator (Global interpretation), creates the sink channel and provides it to the Enabler.

These core functions may be allocated on multiple hosts and interact with one another through the ESB.

Listing 9.1: Drools latency rule example

```
rule "transitionDuration"
  no-loop
  salience 999
  dialect "java"
  when
    $aEvent : SimpleEvent(this.data == "transitionToMonitor");
    $bEvent : SimpleEvent(this.getConnectorID == $aEvent.getConnectorID,
              this.getConnectorInstanceID == $aEvent.getConnectorInstanceID,
              this.getConnectorInstanceExecutionID == $aEvent.getConnectorInstanceExecutionID,
              this after $aEvent);
  then
    retract( $aEvent );
    retract( $bEvent );
    ResponseDispatcher.NotifyMe(drools.getRule().getName(),"enablerName",
                                DroolsUtils.latency($aEvent.getTimestamp(),
                                $bEvent.getTimestamp()));
end
```

In listing 9.1, an example of latency rule expressed using Drools Fusion is given. The rule measures the latency between the 'transitionToMonitor' event and any event that occurs after it.

As shown, using the object SimpleEvent, an implementation of ConnectBaseEvent, it is possible to identify the CONNECTOr, the instance and the iteration of a specific instance of it.

9.4 Runtime Operation

GLIMPSE is an 'always-on' component of the CONNECT architecture. It starts its activity when it receives a message (JMS Message containing ConnectBaseEvent object on the payload) on the service channel.

Figure 9.2 represents a sequence diagram of a basic GLIMPSE interaction.

To communicate with the Monitor, the Enablers need only be able to send/receive JMS Messages. This requirement can be satisfied from heterogeneous kind of devices; in fact we have JMS implementation in Java, C Sharp, C++ and even on iPhone using MQTT broker.

When the Manager starts, the Monitoring Bus (ActiveMQ and ServiceMix4) is already working; Manager starts listening on service channel for Enablers requests (see message 1 in figure 9.2), fires up the CEP Evaluator with an empty Knowledge Base, that defines the behaviour of the engine (see message 2 in Figure 9.2).

The Enabler sends a request on serviceTopic (see message 3 in Figure 9.2). The data field of the ConnectBaseEvent contained into the payload of the JMS Message (the specific type of the JMSMessage is ObjectMessage), is an XML String, that defines the operation to do on the KnowledgeBase of the rule engine.

These operations express the action to take with the Drools rule expressed into the RuleBody field of the XML contained in the ConnectBaseEvent data field. In listing 9.2 the XMLSchema of these operations.

The Manager component that receives an instance of this XMLSchema, takes the action expressed (insert, delete, start, stop, restart) by inserting, deleting,[..], on the CEP Knowledge Base. This message will be notified to the Manager (see message 4 in figure 9.2). The rule may include parameters like: *window-time to evaluate, pattern to match, events to match*. When receives a request, the Manager creates the Sink channel where to write the results (see message 5 in figure 9.2).

The Manager sends to the Enablers the channel name where to find response to the submitted request (see message 6 in figure 9.2), the message is notified to the Enablers (see message 7 in figure 9.2). The Manager now update settings of the Complex Event Processor with the evaluation request received from

Listing 9.2: XML Schema for evaluation request

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule"
        xmlns:tns="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule"
        elementFormDefault="qualified">
    <element name="ComplexEventRuleActionList" type="tns:ComplexEventRuleActionType" />
    <complexType name="ComplexEventRuleActionType">
        <sequence>
                <element name="Insert" type="tns:ComplexEventRuleType"
                        maxOccurs="unbounded" minOccurs="0">
                </element>
                <element name="Delete" type="tns:ComplexEventRuleType"
                        maxOccurs="unbounded" minOccurs="0">
                </element>
                <element name="Start" type="tns:ComplexEventRuleType"
                        maxOccurs="unbounded" minOccurs="0">
                </element>
                <element name="Stop" type="tns:ComplexEventRuleType"
                        maxOccurs="unbounded" minOccurs="0">
                </element>
                <element name="Restart" type="tns:ComplexEventRuleType"
                        maxOccurs="unbounded" minOccurs="0">
                </element>
        </sequence>
    </complexType>

    <complexType name="ComplexEventRuleType">
        <sequence>
                <element name="RuleName" type="string" maxOccurs="1" minOccurs="1" />
                <element name="RuleBody" type="string" maxOccurs="1" minOccurs="0" />
        </sequence>
        <attribute name="RuleType" type="string" />
    </complexType>
</schema>
```

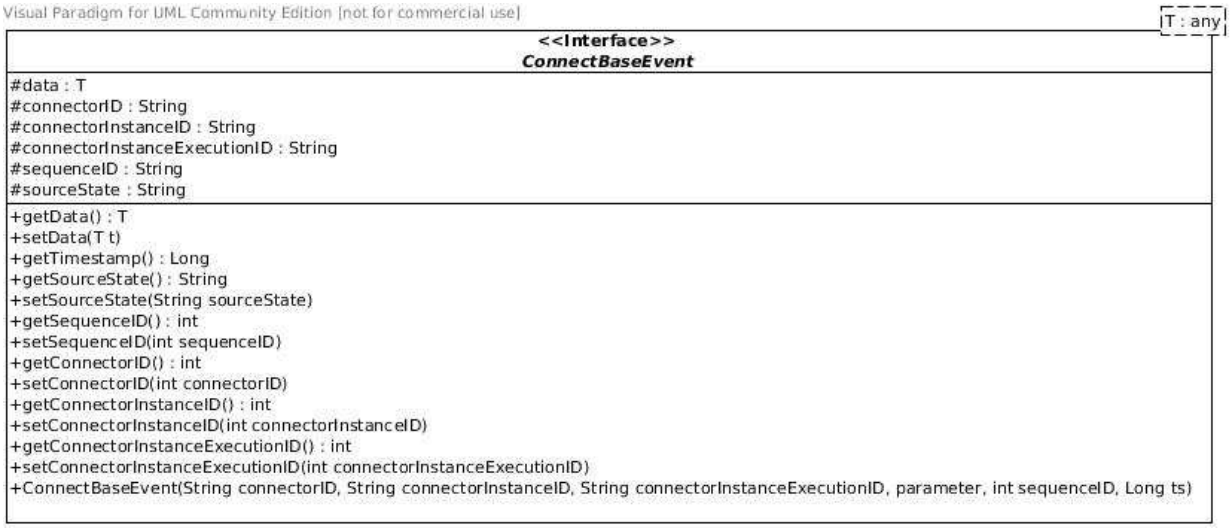

Figure 9.3: CONNECTBaseEvent Interface

the Enablers and with the channel where to write evaluation results (see message 8 in figure 9.2). The Enablers subscribes the Sink channel (see message 9 in figure 9.2). The CEP subscribes the Source channel to receives messages coming from the Probes (see message 10 in figure 9.2), that may be instructed by Manager to apply a filter for reducing amount of raw data sents (see message 11 in figure 9.2).

The Probes, when the CONNECTor starts to works, sends primitive events on the Source channel (see message 12 in figure 9.2), these messages are notified to the CEP Evaluator (see message 13 in figure 9.2), that starts analyze it (see message 14 in figure 9.2) and publish the evaluation on the Sink channel (see message 15 in figure 9.2).

The evaluation is notified to the Enablers (see message 16 in figure 9.2).

Structure of events The events sent by Probes, in the prototype, are an implementation of ConnectBaseEvent. An event in the current system is an abstraction of the transition between states of an LTS machine.

A generic ConnectBaseEvent is composed by:

- a Timestamp;
- a Data field where the probes will put the event payload;
- a ConnectorID, the identifier of the Probe that sent it;
- a ConnectorInstanceID, the identifier of the instance of the CONNECTor that is running;
- a ConnectorInstanceExecutionID, the identifier of the specific execution of an instance of a CONNECTor;
- a SequenceID, may be useful for the CEP to reconstruct strict sequencing between ConnectBaseEvents sent from the same source;
- a SourceState, indicates the state from where the transition started.

9.5 Related Work

Monitoring is recognized as a key functionality in many different types of systems, spanning web services applications, grid environments [108], networks management mechanisms. A first attempt to overview

the most important problems and issues about monitoring in distributed systems is [77]. Due to different aspects and activities of monitoring, research on it appears fragmented. The many works on monitoring are hard to compare as most of them focus only on some of the aspects of monitoring.

The works more related to our proposal are those approaching the definition of a monitoring architecture [90, 63]. In particular, [90] presents an extended event-based middleware with complex event processing capabilities on distributed systems. Similar to GLIMPSE this work adopts a publish/subscribe infrastructure but it is mainly focused on the definition of a complex-event specification language. The aim of GLIMPSE is to give a more general and flexible monitoring infrastructure for achieving a better interpretability with many possible heterogeneous systems.

Another monitoring architecture for distributed systems management is presented in [63]. Differently from GLIMPSE, this architecture employs a hierarchical and layered event filtering approach. The goal of the authors is to improve monitoring scalability and performance for large-scale distributed systems, minimizing the monitoring intrusiveness.

A very attractive research field is the event-based modeling and computing. Many works focus on the definition of expressive complex event specification languages [78, 37, 43]. Among these languages, GEM [78] is a generalized and interpreted event monitoring language. It is rule-based (similar to other event-condition-action approaches) and also provides a tree-bases detection algorithm taking into account communication delay. Also the Snoop language [37] follows an event-condition-action approach supporting temporal and composite events specification but it is especially developed for active databases. A more recent formally defined specification language is TESLA [43]. It has a simple syntax and a semantics based on a first order temporal logic. The authors of [43] also provide an efficient event detection algorithm by translating TESLA rules into automata. Some existing open-source event processing engines are Drools Fusion [2] and Esper [3]. They can fully be embedded in existing Java architectures and provide efficient rule processing mechanisms. In our prototype we used Drools because ServiceMix offers it as business rule engine.

Monitoring can be used to observe functional and non functional properties. An approach to perform automated and distributed monitoring for guarantee Service Level Agreements (SLA) in the web services scenario is presented in [94]. It gives a flexible but precise formalization of SLAs and provides an engine to automate the monitoring of these SLAs. Other monitoring frameworks exist that address the monitoring of performance, in the context of system management. Among them, there are Nagios [19] and Ganglia [83]: the first offers a monitoring infrastructure to support the management of IT systems spanning network, OS, applications; the latter is especially dedicated for high-performance computing and is used in large clusters, focusing on scalability through a layered architecture.

10 Conclusion and Future Work

During Year 2, CONNECT has made significant progress and achieved some breakthroughs towards realizing a learning enabler that can generate useful models of realistic components in network systems. Important advances have been made concerning efficient exploration of component behavior, and concerning extensions of learning technology to rich models combining control, data, and nonfunctional aspects, as well as to extend the framework with monitoring technology.

During Y3, a major overall goal is to leverage the advances in Y2 to actually realize a learning enabler, and evaluate it on realistic applications. Directions of work that should be pursued include:

A Learning Enabler for Rich Component Models The techniques developed during Y2 for learning component models with data should be matured during Y3. This includes to make such models truly scalabe, to adapt them to different classes of systems (web services, middleware), each having a different form of usage of data in combination with control, and to develop and implement techniques for learning models of such components. It also includes further developing the approach described in Chapter 7 towards learning that is specifically tailored for use by the synthesis enabler, and to further develop the approach to handling non-functional properties.

Development of LearnLib Theoretical progress on core learning methods was backed up by the practical implementation of those methods within the framework of LearnLib, our central tool for automata learning. This enabled the evaluation of the conceived approaches in terms of implementability and performance, as well as the conduction of first case studies.

About monitoring During Y2 the we focused on making a generic and flexible monitoring infrastructure using a model-driven approach and advanced messaging technologies. These efforts should be improved in Y3 by integrating the existing monitoring infrastructure with the learning enabler. This will be done with the aim of providing a semantic anomaly detection system able to infer invariants on online data analysis.

We are also quite active in efforts to improve the state of the art of practical learning. A previous effort in this direction was the ZULU challenge, which brought different communities working on learning together, allowing them to mutually benefit from each other's advances. We intend to continue this effort towards establishing a community of researchers and practitioners. One can already see a significant activity concerning learning in leading conferences, and several workshops have been organized on this topic (e.g., at ISoLA 2010, organized by CONNECT partner TUDo). We also envisage to support community-wide efforts towards issues like

- establishing scenarios for realistic case studies,
- developing methods for making automata learning scalable,
- extending active learning to a range of different kinds of models, such as I/O-automata [10],
- providing technology to support the system interaction required for learning, and
- generalizing the notion of learning to capture new kinds of system behavior.

We hope that our efforts will develop as a platform for synergetic cooperation, that the organized challenges will help to identify the strength and weaknesses of the various solutions and their mutual interplay, and that the possibility for easy experimentation, e.g. using LearnLib, will make automata learning a convenient tool in many application contexts.

Bibliography

[1] Activemq: A complete message broker. http://activemq.apache.org.

[2] Drools fusion: Complex event processor. http://www.jboss.org/drools/drools-fusion.html.

[3] Esper: Event stream and complex event processing for java. http://www.espertech.com/products/esper.php.

[4] Jbi: Java business integration. http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html.

[5] Rpc: Model for programming in a distributed computing environment. http://msdn.microsoft.com/en-us/library/ms691207(VS.85).aspx.

[6] Ruleml: The rule markup initiative. http://ruleml.org.

[7] Servicemix: an open source esb. http://servicemix.apache.org/home.html.

[8] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proc. ICTSS, 22nd IFIP WG 6.1 International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, 2010*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[9] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. In *Proc. ISoLA, 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation 18-20 October 2010 - Amirandes, Heraclion, Crete*, volume 6415 of *LNCS*. Springer, 2010.

[10] F. Aarts and F. Vaandrager. Learning I/O Automata. In *Proc. CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.

[11] P. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. Simulation subsumption in ramsey-based büchi automata universality and inclusion testing. In *Proc. CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2010.

[12] P. Abdulla, Y.-F. Chen, L. Holik, R. Mayr, and T. Vojnar. When simulation meets antichains (on checking language inclusion of NFA's). In *Proc. TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 158–174. Springer.

[13] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web service semantics-WSDL-S. *W3C member submission*, 7, 2005.

[14] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126:183–235, April 1994.

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

[16] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[17] J. L. Balcázar, J. Díaz, and R. Gavaldà. Algorithms for Learning Finite Automata from Queries: A Unified View. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.

[18] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 1–3, 2002.

[19] W. Barth. *Nagios. System and Network Monitoring*. No Starch Press, u.s. ed edition, 2006.

[20] M. Benedikt, C. Ley, and G. Puppis. What you must remember when processing data words. In *Proc. 4th Alberto Mendelzon Int. Workshop on Foundations of Data Management, Buenos Aires, Argentina, May 17-20, 2010*, volume 619 of *CEUR Workshop Proceedings*.

[21] A. Bennaceur, G. S. Blair, F. Chauvel, N. Georgantas, P. Grace, F. Howar, P. Inverardi, V. Issarny, M. Paolucci, A. Pathak, R. Spalazzese, B. Steffen, and B. Souville. Towards an Architecture for Runtime Interoperability. In *Proceedings of ISoLA 2010*, 2010.

[22] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the Correspondence Between Conformance Testing and Regular Inference. In M. Cerioli, editor, *Proc. FASE '05,* 8^{th} *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer Verlag, April 4-8 2005.

[23] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines with Parameters. In L. Baresi and R. Heckel, editors, *Proc. FASE '10,* 13^{th} *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer Verlag, 2006.

[24] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In J. L. Fiadeiro and P. Inverardi, editors, *Proc. FASE '08,* 11^{th} *Int. Conf. on Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer Verlag, 2008.

[25] A. Bertolino, G. De Angelis, A. Sabetta, and S. Elbaum. Scaling up SLA monitoring in pervasive environments. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, ESSPE '07, pages 65–68, New York, NY, USA, 2007. ACM.

[26] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 141–150, New York, NY, USA, 2009. ACM.

[27] M. Bielecki, J. Hidders, J. Paredaens, J. Tyszkiewicz, and J. V. den Bussche. Navigating with a browser. In *Proc. ICALP '2002,* 29^{th} *International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 764–775. Springer, 2002.

[28] H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411:702–715, January 2010.

[29] T. Bohlin, B. Jonsson, and S. Soleimanifard. Inferring compact models of communication protocol entities. In *Proc. ISoLA, 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation 18-20 October 2010 - Amirandes, Heraclion, Crete*, volume 6415 of *Lecture Notes in Computer Science*, pages 658–672. Springer, 2010.

[30] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proc. LICS' 06* 21^{nd} *IEEE Int. Symp. on Logic in Computer Science*, pages 7–16, Washington, DC, USA, 2006. IEEE Computer Society.

[31] P. Bouyer. A logical characterization of data languages. *Inf. Process. Lett.*, 84:75–85, October 2002.

[32] P. Bouyer, A. Petit, and D. Thérien. An algebraic approach to data languages and timed languages. *Information and Computation*, 182(2):137–162, 2003.

[33] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems:*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[34] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04: 26th Int. Conf. on Software Enginering*, May 2004.

[35] S. Callanan, D. Dean, M. Gorbovitski, R. Grosu, J. Seyster, S. Smolka, S. Stoller, and E. Zadok. Software monitoring with bounded overhead. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[36] T. D. Chair of Programming Systems, Department of Computer Science. RERS - A Challenge In Active Learning. http://leo.cs.tu-dortmund.de:8100/. Version from 20.06.2010.

[37] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[38] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. on Software Engineering*, 4(3):178–187, May 1978.

[39] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[40] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03, 9^{th} 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.

[41] D. Combe, C. de la Higuera, and J.-C. Janodet. Zulu: an Interactive Learning Competition. In *Proceedings of FSMNLP 2009*, 2010. to appear.

[42] D. Combe, C. de la Higuera, J.-C. Janodet, and M. Ponge. Zulu - Active learning from queries competition. http://labh-curien.univ-st-etienne.fr/zulu/index.php. Version from 01.08.2010.

[43] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 50–61, New York, NY, USA, 2010. ACM.

[44] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 69(1-3):35–45, 2007.

[45] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.

[46] N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theoretical Computer Science*, 306(1-3):155–175, 2003.

[47] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite State Models. *IEEE Trans. on Software Engineering*, 17(6):591–603, 1991.

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

[49] O. Grinchtein. *Learning of Timed Systems*. PhD thesis, Dept. of IT, Uppsala University, Sweden, 2008.

[50] O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. In *Proceedings of the Joint Conferences FORMATS and FTRTFT*, volume 3253 of *LNCS*, pages 379–396, Sept. 2004.

[51] O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. *Theor. Comput. Sci.*, 411(47):4029–4054, 2010.

[52] 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.

[53] R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, volume 5047 of *Lecture Notes in Computer Science*, pages 216–233, 2008.

[54] 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.

[55] D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and System Modeling*, 2(2):82–107, 2003.

[56] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[57] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[58] F. Howar, B. Jonsson, M. Merten, B. Steffen, and S. Cassel. On Handling Data in Automata Learning - Considerations from the CONNECT Perspective. In Margaria and Steffen [81], pages 221–235.

[59] F. Howar, B. Steffen, and M. Merten. From ZULU to RERS - lessons learned in the ZULU challenge. In Margaria and Steffen [80], pages 687–704.

[60] F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *Twelfth International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011.

[61] A. Huima. Implementing conformiq qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Proc. TestCom/FATES, Tallinn, Estonia, June, 2007*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12, 2007.

[62] H. Hungar, O. Niese, and B. Steffen. Domain-Specific Optimization in Automata Learning. In W. A. H. Jr. and F. Somenzi, editors, *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, July 2003.

[63] E. A.-S. Hussein and Abdel-Wahab. Hifi: A new monitoring architecture for distributed systems management. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 171–178, Washington, DC, USA, 1999. IEEE Computer Society.

[64] P. Inverardi, V. Issarny, and R. Spalazzese. A Theory of Mediators for Eternal Connectors. In *Proceedings of ISoLA 2010*, 2010.

[65] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2):272–302, Dec. 1993.

[66] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987.

[67] G. Jung, T. Margaria, C. Wagner, and M. Bakera. Formalizing a Methodology for Design- and Runtime Self-Healing. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on*, 0:106–115, 2010.

[68] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[69] P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[70] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. Cambridge, MA, USA, 1994.

[71] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *Proc. ICGI-98, 4th Int. Coll. Grammatical Inference, Ames, Iowa, USA*, volume 1433 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[72] LearnLib - a framework for automata learning. http://www.learnlib.de, 2011.

[73] 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.

[74] D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. In *Proc. ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium*, pages 387–396. ACM, 2010.

[75] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. ICSE'08: 30th Int. Conf. on Software Enginering*, pages 501–510, 2008.

[76] O. Maler and A. Pnueli. On the Learnability of Infinitary Regular Sets. *Information and Computation*, 118(2):316–326, 1995.

[77] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems. *Network and distributed systems management*, pages 303–347, 1994.

[78] M. Mansouri-Samani and M. Sloman. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[79] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 95–100, Washington, DC, USA, 2004. IEEE Computer Society.

[80] T. Margaria and B. Steffen, editors. *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*. Springer, 2010.

[81] T. Margaria and B. Steffen, editors. *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II*, volume 6416 of *Lecture Notes in Computer Science*. Springer, 2010.

[82] L. Mariani and M. Pezzé. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, September/October 2007.

[83] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[84] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next Generation LearnLib. In *Seventeenth International Conference on TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS*, 2011.

[85] A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[86] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.

[87] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[88] 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.

[89] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. on Software Engineering*, 30(1):29–42, 2004.

[90] P. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44 – 55, jan. 2004.

[91] H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. 11(4):307–324, 2009.

[92] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. 11(5):393–407, 2009.

[93] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[94] A. Sahai, V. Machiraju, M. Sayal, A. van Moorsel, and F. Casati. Automated sla monitoring for web services. In M. Feridun, P. Kropf, and G. Babin, editors, *Management Technologies for E-Commerce and E-Business Applications*, volume 2506 of *LNCS*, pages 28–41. Springer, 2002.

[95] H. Sakamoto. Learning simple deterministic finite-memory automata. In *ALT '97: Proc. 8th International Conference on Algorithmic Learning Theory, Sendai, Japan.*, volume 1316 of *Lecture Notes in Computer Science*, pages 416–431. Springer Verlag, Oct. 1997.

[96] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31, 2006.

[97] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL: Proc. 20th Int. Workshop on Ccomputer Science Logic, Szeged, Hungary*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006.

[98] M. Shahbaz and R. Groz. Inferring Mealy Machines. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, pages 207–222, Berlin, Heidelberg, 2009. Springer Verlag.

[99] 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.

[100] G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proc. ICDCS'07, 27th IEEE Int. Conf. on Distributed Computing Systems, Toronto, Ontario*. IEEE Computer Society, 2007.

[101] B. Steffen, F. Howar, M. Merten, and T. Margaria. *FMICS Handbook*, chapter Practical Aspects Of Active Learning. Wiley, to appear in 2010.

[102] B. Steffen, T. Margaria, H. Raffelt, and O. Niese. Efficient test-based model generation of legacy systems. In *HLDVT'04: Proc. of the 9th IEEE Int. Workshop on High Level Design Validation and Test*, pages 95–100, Sonoma (CA), USA, November 2004. IEEE Computer Society Press.

[103] B. Steffen and O. Rüthing. Quality engineering: Leveraging heterogeneous information. In *Twelfth International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 23–37, 2011.

[104] The XMPP Standards Foundation. http://xmpp.org/about-xmpp/, 2011.

[105] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. on Software Engineering*, 29(2):99–115, Feb. 2003.

[106] S. Verwer, M. de Weerdt, and C. Witteveen. One-clock deterministic timed automata are efficiently identifiable in the limit. In *LATA 2009: Proc. 3rd Int. Conf. Language and Automata Theory and Applications, Tarragona, Spain*, volume 5457 of *Lecture Notes in Computer Science*, pages 740–751. Springer, 2009.

[107] P. Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc.* 13^{th} *ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.

[108] S. Zanikolas and R. Sakellariou. A taxonomy of grid monitoring systems. *FGCS Journal*, 21:163–188, 2005.